8-4-2011

A Method for detection and quantification of building damage using post-disaster LiDAR data

Richard Labiak

A METHOD FOR DETECTION AND QUANTIFICATION OF BUILDING DAMAGE USING POST-DISASTER LIDAR DATA

by

Richard C. Labiak

Bachelor of Science, Electrical Engineering

United States Air Force Academy, 2006

A thesis submitted in partial fulfillment of the

requirements for the degree of Master of Science

in the Chester F. Carlson Center for Imaging Science

Rochester Institute of Technology

August 4, 2011

Signature of the Author_____

Accepted by_____
Director, M.S. Degree Program Date

CHESTER F. CARLSON CENTER FOR IMAGING SCIENCE

ROCHESTER INSTITUTE OF TECHNOLOGY

ROCHESTER, NEW YORK

CERTIFICATE OF APPROVAL

M.S. DEGREE THESIS

The M.S. Degree Thesis of Richard C. Labiak
has been examined and approved by the
thesis committee as satisfactory for the
thesis requirement for the
M.S. degree in Imaging Science

_____
Dr. Jan A.N. van Aardt, Thesis Advisor

_____
Dr. David W. Messinger

_____
Dr. Harvey E. Rhody

_____
Date

THESIS RELEASE PERMISSION

CHESTER F. CARLSON CENTER FOR IMAGING SCIENCE

ROCHESTER INSTITUTE OF TECHNOLOGY

Title of Thesis:

A Method for Detection and Quantification of Building Damage Using Post-Disaster LiDAR Data

I, Richard C. Labiak, hereby grant permission to the Rochester Institute of Technology to reproduce my print thesis or dissertation in whole or in part. Any reproduction will not be for commercial use or profit.

Signature of Author: _____ Date: _____

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

Abstract

There is a growing need for rapid and accurate damage assessment following natural disasters, terrorist attacks, and other crisis situations. The use of light detection and ranging (LiDAR) data to detect and quantify building damage following a natural disaster was investigated in this research. Using LiDAR data collected by the Rochester Institute of Technology (RIT) just days after the January 12, 2010 Haiti earthquake, a set of processes was developed for extracting buildings in urban environments and assessing structural damage. Building points were separated from the rest of the point cloud using a combination of point classification techniques involving height, intensity, and multiple return information, as well as thresholding and morphological filtering operations. Damage was detected by measuring the deviation between building roof points and dominant planes found using a normal vector and height variance approach. The devised algorithms were incorporated into a Matlab graphical user interface (GUI), which guided the workflow and allowed for user interaction. The semi-autonomous tool ingests a discrete-return LiDAR point cloud of a post-disaster scene, and outputs a building damage map highlighting damaged and collapsed buildings.

The entire approach was demonstrated on a set of six validation sites, carefully selected from the Haiti LiDAR data. A combined 85.6% of the truth buildings in all of the sites were detected, with a standard deviation of 15.3%. Damage classification results were evaluated against the Global Earth Observation – Catastrophe Assessment Network (GEO-CAN) and Earthquake Engineering Field Investigation Team (EEFIT) truth assessments. The combined overall classification accuracy for all six sites was 68.3%, with a standard deviation of 9.6%. Results were impacted by imperfect validation data, inclusion of non-building points, and very diverse environments, e.g., varying building types, sizes, and densities. Nevertheless, the processes exhibited significant potential for detecting buildings and assessing building-level damage.

Acknowledgments

This thesis would not have been possible without the help and support of many individuals. First and foremost, I would like to thank my advisor, Dr. Jan van Aardt, for guiding me through this process and always having my best interests in mind. He challenged me to think critically and taught me to be confident in my work. Thank you Jan for being so responsive and always having the time to meet with me. I would also like to thank my other committee members, Dr. David Messinger and Dr. Harvey Rhody, for helping me stay on track and for providing valuable insight.

This project required understanding and access to the data collected during the Haiti campaign. To this end, Jason Faulring and Steve Cavilia were invaluable resources and I really appreciate all of their help. I would also like to thank Don McKeown for providing the EEFIT truth data, and Cindy Schultz for all of her administrative support. I would like to acknowledge all of my Imaging Science instructors, for not only giving me the knowledge base to complete this research, but also for developing my interest in the field. A special thanks goes out to my fellow Air Force and LiDAR students, who have been great officemates and friends.

Finally, I would like to thank my parents and close friends. I truly appreciate the constant support you guys have given me throughout the past two years. Thanks for being so patient with me, and know that your encouragement and love did not go unnoticed.

Table of Contents

Page

List of Figures

1 Introduction

On January 12, 2010 at 4:53 PM local time, a magnitude 7.0 M_W earthquake struck the Republic of Haiti. This natural disaster, one of the most destructive in history, killed an estimated 230,000 people, injured 300,000, and displaced 1.6 million from their damaged or destroyed homes (USAID, 2010). The epicenter of the earthquake was located 25 km southwest of Port-au-Prince, Haiti's largest city and densely populated capital. At the time of the disaster, roughly 2 million people lived within the zone that suffered heavy to moderate structural damage (CIA, 2011). According to the United States Geological Survey (USGS), 97,294 houses were destroyed and 188,383 were damaged in Port-au-Prince and in much of southern Haiti (USGSa, 2011). In the days following the earthquake, individuals and organizations from around the world descended upon Haiti to help search for survivors, provide food, water, medicine, and shelter, and start the rebuilding effort.

Logistical challenges quickly surfaced as international humanitarian relief and military aid poured into the Caribbean country. Toussaint Louverture International Airport, Port-au-Prince's only international airfield, had just a single runway, single taxiway, and a small, crowded ramp (Jones, 2011). The city's main seaport was closed after severe damage to the docks and its one major gantry crane. In addition, many of the roads were impassable, either directly damaged by the earthquake or blocked by rubble (Davidson and Smith, 2011). The sheer scale of the disaster presented rescuers with a seemingly overwhelming task, and a need quickly arose for the ability to provide timely damage assessment information to aid in disaster management.

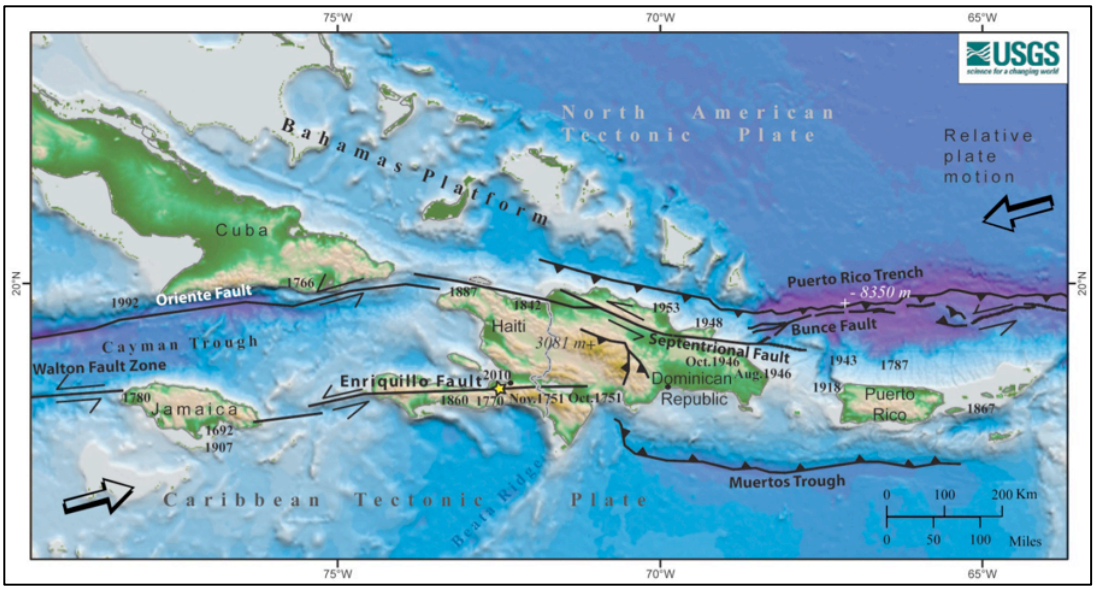


Figure 1. Topographic map showing the northern Caribbean plate boundary. The January 12, 2010 Haiti earthquake, marked by a star, occurred along the Enriquillo Fault (Eberhard *et al.*, 2010).

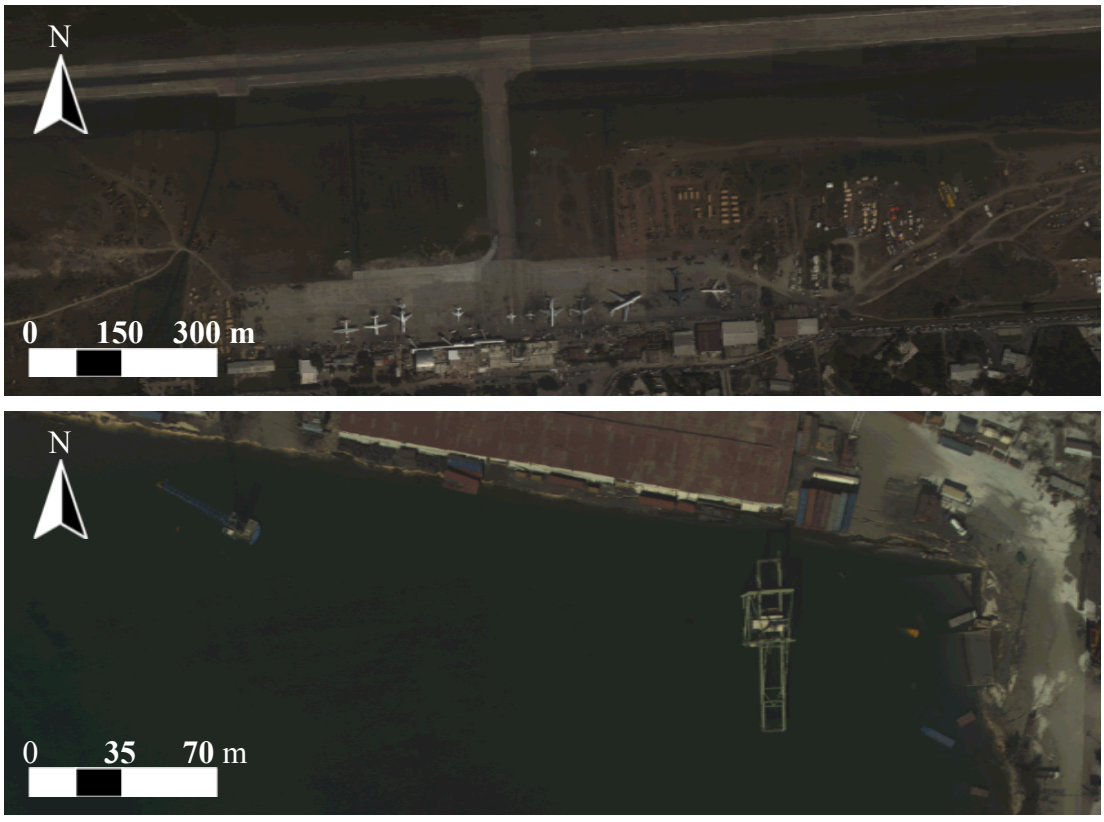


Figure 2. Logistical challenges presented relief workers with a difficult task. *Above:* Toussaint Louverture International Airport in Port-au-Prince with one runway and a small cargo ramp. *Below:* The main seaport in Port-au-Prince was closed after the docks, a gantry crane, and some shipping containers collapsed into the water (Wildfire Airborne Sensing Program (WASP) imagery).

According to Ronald Eguchi, President and CEO of ImageCat, Inc., the Haiti earthquake was one of the first events where remote sensing technology was "embraced at such a large scale in a real operational sense" (Eguchi *et al.*, 2010). Starting within days of the disaster, vast amounts of high-resolution satellite and aerial optical imagery and Light Detection and Ranging (LiDAR) data were collected on a daily basis. ImageCat, the World Bank's Global Facility for Disaster Reduction and Recovery (GFDRR), the Rochester Institute of Technology (RIT), the Earthquake Engineering Research Institute (EERI), and the Multidisciplinary Center for Earthquake Engineering Research (MCEER) were among the organizations that used these data to guide damage assessment and rescue and recovery efforts (Eberhard *et al.*, 2010).

ImageCat, in partnership with many of the organizations listed above, formed a worldwide network of engineers and scientists tasked with analyzing Haiti imagery and performing damage assessment on more than 30,000 buildings. The Global Earth Observation – Catastrophe Assessment Network (GEO-CAN), consisting of over 600 volunteers at its height, manually identified damaged buildings by comparing a combination of 50 cm resolution Geo-Eye-1 imagery and 15 cm resolution Google and Wildfire Airborne Sensing Program (WASP) imagery with imagery collected before the disaster (Bevington *et al.*, 2010). GEO-CAN officially started on January 21, 2010, though thermal IR and LiDAR were not used until March 5, 2010, when the third phase of the effort got underway to identify debris, liquefaction, and investigate thermal and geological anomalies caused by the earthquake (EERI, 2010). An example GEO-CAN damage map is shown in Figure 3.

In addition to the GEO-CAN campaign, the National Geospatial-Intelligence Agency (NGA), the German Space Agency (DLR), Information Technology for Humanitarian Assistance Cooperation and Action (ITHACA), and the United Nations Operational Satellite Applications Program (UNOSAT) were among numerous other agencies that either used or made available imagery to aid in recovery and rebuilding in Haiti (Eberhard *et al.*, 2010). Figure 4 shows a damage assessment product that combines information from these agencies. The map not only highlights areas of moderate to catastrophic damage in red, but it also displays the location and condition of field medical sites (ReliefWeb, 2010).

Since the availability of high quality remote sensing data is no longer a limiting factor in disaster response, focus is now being directed towards developing useful information products that can be derived from the data. Eguchi states, "While the area of remotely-sensed damage assessment took a quantum step forward in the Haiti campaign, there is still significant room for improvement" (Eguchi *et al.*, 2010). Much of the damage assessment in Haiti took days to weeks to perform, mostly due to the intense and manual nature of the work. The availability of a damage map produced in near real-time would help satisfy the never-ending need to reduce the response phase in the disaster management cycle. Creating a damage map within hours of a natural disaster is not a trivial task, but this research details an effort using LiDAR technology and limited human interaction to realize this goal.

Figure 3. Example GEO-CAN damage assessment map of Port-au-Prince. The buildings outlined in red were manually classified as being damaged by a network of over 600 skilled volunteers (Eberhard *et al.*, 2010).

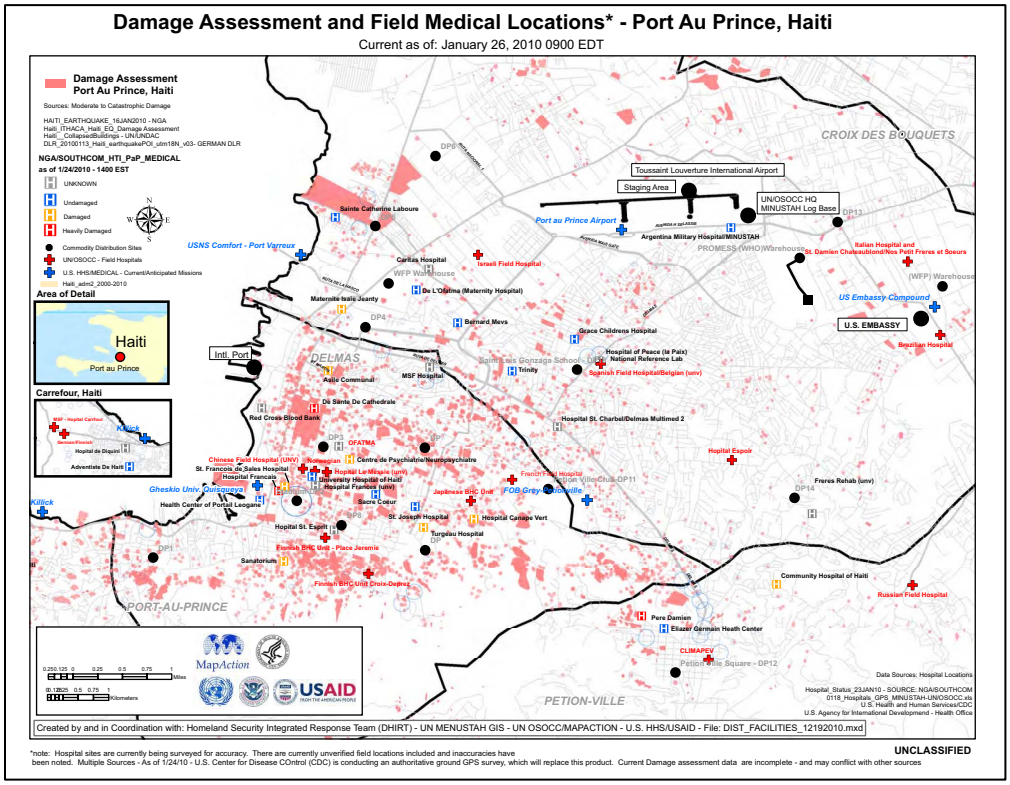


Figure 4. Map showing damage assessment and location and condition of field hospital sites in Port-au-Prince. The data are current as of January 26, 2010, two weeks following the magnitude 7.0 earthquake (ReliefWeb, 2010).

2 Project Overview

2.1 Research Questions

Immediately following the occurrence of a natural disaster, a quick survey of the damage is often more important than a detailed damage assessment (van den Broek *et al.*, 2009). A rapid damage map can help to determine where first responders and relief workers should be sent and how to prioritize their efforts. It informs those on the ground about the relative safety of areas and structures, and in the case of Haiti, where displaced personnel camps should be constructed. On a strategic level, initial damage estimates can help to budget and allocate relief and recovery funds, and highlight critical areas for future data collection and detailed damage assessment.

In this project, the use of LiDAR data to detect and quantify building damage following a natural disaster was investigated. Using only LiDAR data, and only data collected after the earthquake, the goal of this research was to develop processes for rapidly and accurately mapping urban environments and assessing damage. Although algorithm development, testing, and validation were accomplished on LiDAR data of Haiti collected nine days after the earthquake, the intent is that the devised techniques can be extended and available for immediate use in future emergency situations.

2.2 Objectives

The main objective of this research was to develop an end-to-end operational tool that will ingest a discrete-return LiDAR point cloud of a post-disaster scene, and semi-autonomously output a building map showing damaged and collapsed buildings. The tool was developed with the following operational scenario in mind. A natural disaster

occurs and within hours a LiDAR sensor is deployed to scan the affected area. The LiDAR data are retrieved in near real-time and undergo initial point-cloud processing, where the range and orientation of each laser pulse is converted into an *X, Y, Z* position. The resulting point cloud is then used as input to the tool, and within minutes a damage map is created and available to those directing the response activities.

Due to the unknown operating environment and available infrastructure immediately following a disaster, the tool was designed to function within common operating systems on standard hardware. The algorithms were developed and tested using Matlab, but will eventually be implemented in either Java or C++ where processing can be optimized for large data sets. Efficient data structures and parallel computing can also be used, since fast processing of large amounts of data are critical in the disaster response environment.

A number of steps were required to complete this project, as outlined below.

1. **Select several study areas**. Study sites were carefully selected in order to develop robust building segmentation and damage detection algorithms. Multiple study areas were chosen that reflect an assortment of building shapes and sizes, differing levels of building damage, and varying topography.
2. **Preprocess the LiDAR data.** Before buildings can be identified and assessed for damage, the point cloud was processed to accurately reflect height above ground and be free of gross outlier points. Simple height thresholding can remove outlier points, but it can be challenging to remove the effect of the terrain.
3. **Segment buildings.** This research was aimed at enhancing existing algorithms for LiDAR feature classification. Rules were developed to identify building

regions, while removing vegetation, roads, and other unwanted points from the point cloud.

4. **Detect and quantify building damage.** Structure and texture-based metrics were explored at the "roof level" to develop a process for distinguishing between intact and damaged building points.
5. **Validate results.** The accuracy of the resulting building damage map was evaluated using the GEO-CAN assessment as truth data. In addition to reporting the classification results, recommendations were made regarding sensor parameters, and how a future LiDAR instrument should be configured in order to provide a more accurate map.

3 Background

3.1 Chapter Overview

This research leveraged previously published work to segment building regions and evaluate structural damage from a LiDAR point cloud in a post-disaster scenario. Before developing a methodology, the current state of the art is assessed by examining LiDAR technology and understanding how it can be used for emergency response. This is followed by an overview of Digital Terrain Model (DTM) extraction from LiDAR data. Finally, several existing algorithms for building extraction and damage detection are assessed for potential use in this work.

3.2 Light Detection and Ranging (LiDAR) Remote Sensing Systems

In order to understand the issues related to the processing of LiDAR data, and the creation of derived products, it is necessary to first understand the LiDAR mapping process. LiDAR, a type of active remote sensing, uses a pulsed laser to illuminate a field of view and then detects the radiation that is backscattered off an object or medium (Argall and Sica, 2003). Insight into the material properties or position of the scanned object can be gained by analyzing the reflected energy.

LiDAR instruments are typically mounted on small to medium fixed-wing aircraft, though they can be fitted to operate in helicopters, space-borne platforms, or on the ground (Terrapoint, 2008). Airborne systems often use a beam director to scan laser pulses over a strip of terrain orthogonal to the direction of flight (Lohani, 2010). The round trip travel time, t, of the pulse from the transmitter to the interacting object is

measured using an on-board clock, and can be converted into range measurements using the equation

$$D = \frac{c \times t}{2}, \tag{1}$$

where D is the distance from the aircraft to the object and c is the speed of light.

In addition to the transmitter, receiver, and detector, each LiDAR system is also comprised of a global positioning system (GPS) receiver and an inertial navigation system (INS). An example airborne LiDAR system is shown in Figure 5. The precise position of the aircraft at the time of each measurement is determined by GPS, while the INS calculates aircraft roll, pitch, and yaw values. By combining the range measurement, location of the beam director, and aircraft position and orientation information, LiDAR sensors can accurately determine the position of each interaction on the surface of the terrain (Lohani, 2010).

As LiDAR sensor technology has advanced, the laser emission rates have increased from a few pulses per second, to hundreds of thousands of pulses per second. One such technological advancement, termed Multiple Pulses in Air (MPiA), allows LiDAR systems to transmit a second laser pulse prior to the receipt of a previous pulse's return (Roth and Thompson, 2008). This effectively doubles the pulse rate at a given flight altitude. The Leica ALS60, the LiDAR instrument flown by RIT over Haiti, utilized this technology and was operated at pulse rates up to 150,000 Hz.

The resulting range scan output from a LiDAR system is commonly referred to as a "point cloud", since it is comprised of a large number of points, each containing X, Y, Z coordinate (and intensity) information. An example of such a data set is shown in Figure

6, a topographic mapping of a region surrounding Haiti's National Palace. The data collected over Haiti by the Leica ALS60 have a point cloud density of roughly 2-5 points/m^2.

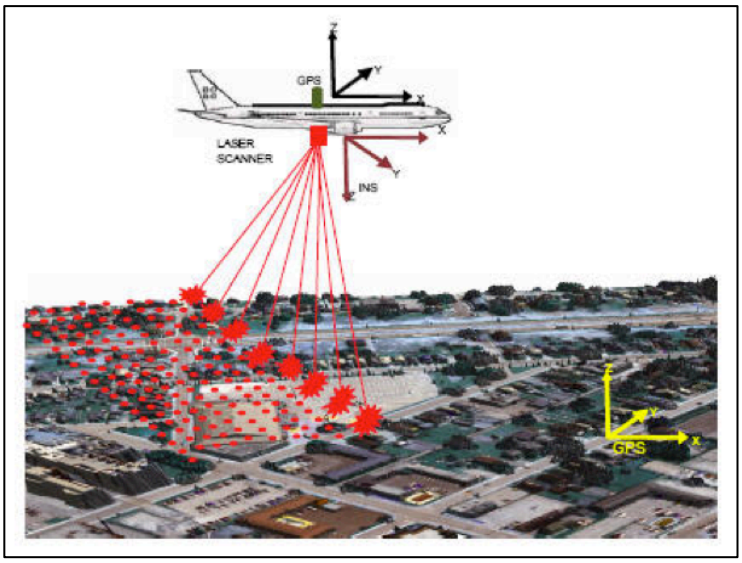


Figure 5. LiDAR instrument mounted to a fixed-wing aircraft. Combining the range, scan angle, laser position from GPS, and laser orientation from INS, accurate *X, Y, Z* ground coordinates can be calculated for each laser pulse (Lohani, 2010).

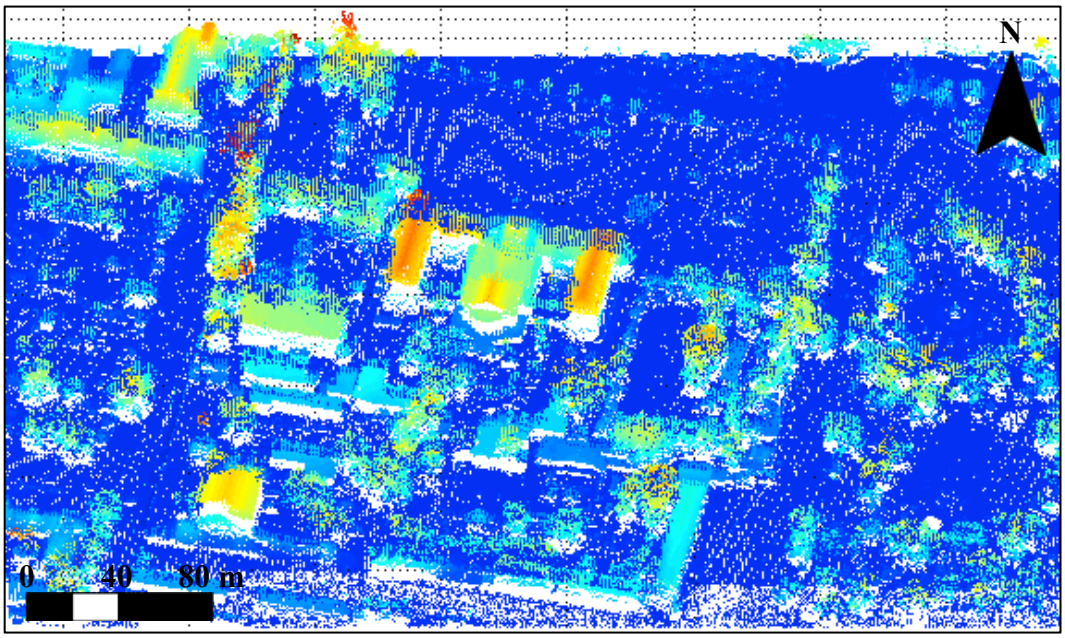


Figure 6. LiDAR point cloud of Haiti's Presidential Palace and surrounding area. The point heights range from 0 m (ground) to almost 27 m, and are colored accordingly. The ground points are shown in royal blue, while the highest points are displayed in red.

In addition to providing position information, LiDAR instruments can also record the intensity of the return signal. Intensity, or the amplitude of the reflected response, is dependent on the emitted wavelength of the laser and reflectance properties of the interacting object (Lach, 2008). The intensity of the return signal will change as surface types and characteristics vary, making the parameter useful in point classification and feature extraction.

Most LiDAR systems can also record multiple returns, often up to seven, from the same outgoing pulse. Multiple returns occur when the beam diameter of the laser pulse does not completely encompass an object. As a result, some of the light continues traveling until it encounters another object, instead of being directly reflected or absorbed. Multiple return information can be used to differentiate points on trees and the edges of rooftops from the rest of the point cloud. Figure 7 shows the point cloud of Haiti's National Palace and surrounding area, colored differently to display height, intensity, and multiple return LiDAR points.

3.3 LiDAR Technology in Disaster Management

For a little over a decade, LiDAR has been used to assist emergency managers in making better decisions. LiDAR has been employed in all stages of the emergency management cycle: mitigation, preparedness, response, and recovery. Mitigation and preparedness activities minimize the impact of future disasters, while response and recovery actions are taken once an event has occurred (Adams, 2006). This section will summarize LiDAR usage in emergency management, considering both pre- and post-event applications.

The identification of probable hazards and assessing risk, are pre-event activities where LiDAR has played an important role. One example is in the creation of landslide inventory maps, which can be used to predict landslide susceptibility. In the Flemish Ardennes, the hilly regions of southern Belgium, these maps are of interest to authorities who are considering imposing land use regulations (Van Den Eeckhaut *et al.*, 2007). Land use regulations and building codes are also directly linked to flood hazard maps, which are created using hydrodynamic-numerical models (Mandlburger *et al.*, 2008). LiDAR is often used to derive precise Digital Terrain Models (DTMs) that serve as a geometric basis for these simulations.

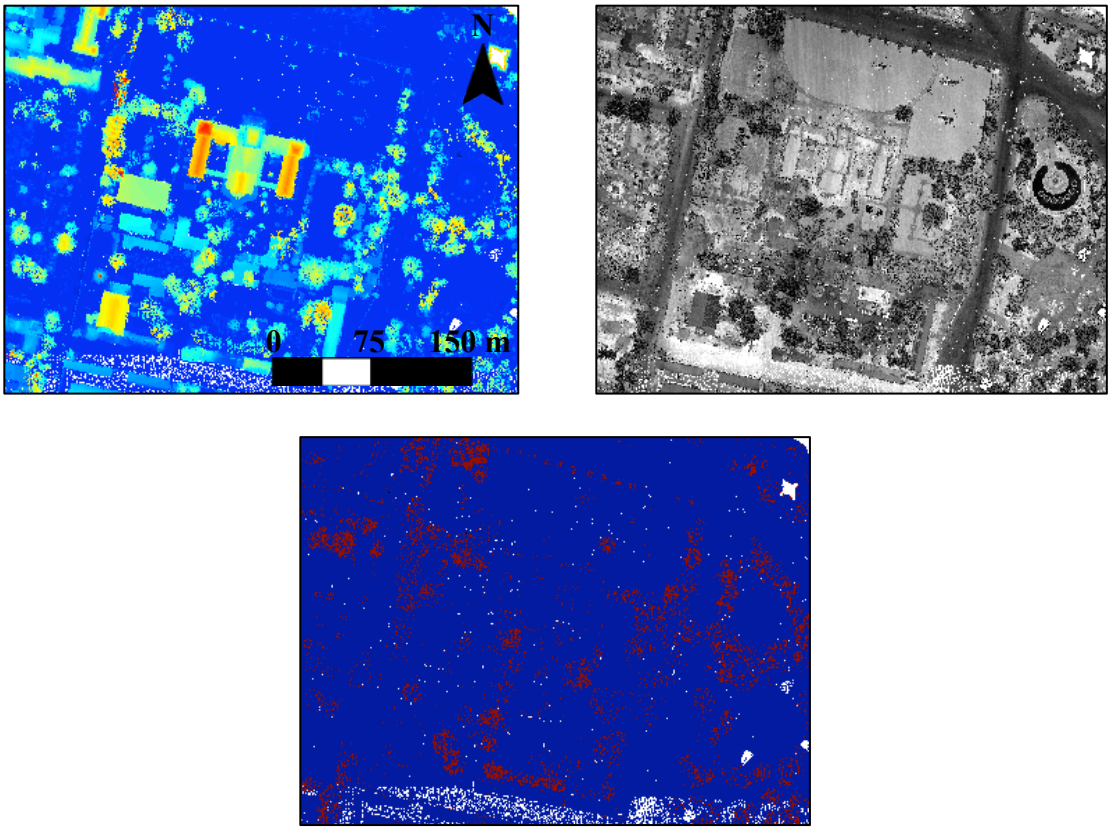


Figure 7. Nadir view of the point cloud of Haiti's Presidential Palace and surrounding area colored to show different information. *Top Left:* Height image with ground points shown in royal blue and the highest points in red. *Top Right:* Intensity image with white representing high intensity or the most reflective objects, and black representing low intensity. *Bottom:* Multiple return image displaying non-first return points in red.

In addition to landslide mapping and river flow modeling, LiDAR can be used to evaluate potential evacuation routes prior to a disaster. This was demonstrated in a study that used 3D data to identify tall objects next to major roads (Laefer and Pradhan, 2006). The research developed an automated method to recognize potential hazards by predicting areas where debris or trees could fall on overhead lines or block the highways. With this knowledge, reliable evacuation routes can be selected before a disaster occurs.

LiDAR data have been used extensively in active disaster response situations as well. LiDAR played a major role in the emergency response effort following the World Trade Center attack on September 11, 2001. For the first couple of weeks after the disaster, EarthData collected LiDAR imagery of Ground Zero on a daily basis. LiDAR enabled emergency managers to assess damage through the smoke and dust, which lasted for days following the terrorist attack (Kwan and Ransberger, 2010). Along with using the 3D elevation data to map the debris pile, fire chiefs, FEMA, and other response personnel used LiDAR difference images to track debris removal and compared surface depressions with the location of hazardous materials and fuel sources (Huyck *et al.*, 2003).

The USGS, NASA, and the U.S. Army Corps of Engineers use laser mapping systems to survey coastal regions before and after hurricanes (USGSb, 2011). The program has documented coastal change in response to Hurricanes Ivan, Katrina, Rita, and Ike, among others. Hurricane Katrina LiDAR data have also been used to detect road obstructions and to analyze the effect that blockages have on traffic patterns and the response time of first responders (Kwan and Ransberger, 2010).

Though there are several remote sensing imaging modalities that could be used for disaster prevention and management, LiDAR's ability to provide a 3D visualization of the disaster area makes it an effective tool for emergency managers. LiDAR has several advantages over traditional photogrammetry, for instance the data can be collected quickly and with a high degree of automation (Baltsavias, 1999). In addition, LiDAR data can be acquired at night or in adverse conditions, such as in poor illumination or through clouds and smoke.

LiDAR, which is well known for being a primary data source for the generation of DTMs and 3D building and city models, is therefore quickly emerging as an effective tool for assessing structural damage. LiDAR data inherently provide precise and reliable elevation information, and thus the capability to identify and measure collapsed and standing buildings.

3.4 Digital Terrain Model (DTM) Extraction from a LiDAR Point Cloud

One objective of this research was to generate a DTM from LiDAR data. A DTM is a digital representation of the bare ground surface, or what is left of the point cloud after buildings, vegetation, and other objects are removed. In order to automatically generate a DTM from LiDAR data, an algorithm must be developed to separate terrain points from non-terrain points (Forlani *et al.*, 2006). Buildings can be extracted from the non-ground points through further processing (Ma, 2005). Though closely related, this section will discuss techniques for classifying points as either ground or non-ground. Feature classification, specifically building extraction approaches, will be explored in Section 3.5.

Many approaches for DTM extraction have been proposed in the literature, though they tend to fall under two broad categories. The first type involves local area processing and the use of morphology and slope-based filters. These techniques are based on the assumptions that the ground is smooth and that ground points are lower than neighboring object points (Ma, 2005). The second strategy attempts to model the ground using a parametric function, and looks for deviations from the predicted surface (Lach, 2008).

Prior to performing point classification, it is typical to resample the irregularly distributed raw point cloud to a uniform 2D grid, or raster image. Though some algorithms use the raw point cloud directly, using raster images is often more advantageous, especially when applying conventional image processing techniques (Lach, 2008). For clarity in the following sections, the raster image of the raw point cloud will be called a Digital Surface Model (DSM), not to be confused with the DTM.

First proposed by Lindenberger (1993), the use of mathematical morphology to filter LiDAR data was one of the original techniques applied to separate terrain from non-terrain points. The technique used a sliding window translated over the DSM, where the pixel at the center of the window was the pixel of interest. The value of each pixel of interest was assigned the minimum value in its neighborhood, or area defined by the window. This minimum filtering is a special case of erosion, where the structuring element, or window, has constant pixel values (Weidner and Förstner, 1995). The next step was to apply a maximum filter to the result, or to replace the pixel of interest with the maximum value in the window. This is a special case of dilation, and in morphology

terms, erosion followed by dilation is referred to as "opening" (Weidner and Förstner, 1995).

A common issue with morphological filtering is choosing the right window size. Too large of a window causes problems if there is high terrain variation, while too small of a window could incorrectly classify building roof points as terrain (Morgan and Habib, 2002). Weidner and Förstner (1995) used a fixed window size chosen to be larger than the largest building. Advanced morphological approaches, such as those proposed by Morgan and Habib (2002), Morgan and Tempfli (2000), and Kilian *et al.* (1996) used windows of variable size, based on a priori knowledge of the building sizes in the scene. In these cases, pixels were assigned certain weights depending on the window size, and thus their chance of being terrain pixels (Kilian *et al.*, 1996). A progressive morphological filter was developed in Zhang *et al.* (2003), in which gradually increasing window sizes, combined with elevation difference thresholds, effectively removed most of the non-ground points. In a random sample of 648 measurements, there were 17 omission and two commission errors made by the filter (Zhang *et al.*, 2003).

Originally proposed by Vosselman (2000), slope based filtering relies on the premise that large height differences between two close points are generally not caused by a steep slope in the terrain. In this method, a point was classified as a non-ground point if the maximum slope of the vectors connecting a point to its neighbors was larger than a predefined threshold (Ma, 2005). However, the technique was limited to terrain with gentle slopes due to the assumption that terrain slopes do not rise above a certain threshold. This limitation was overcome by Sithole (2001) through modifcation of the filter so the threshold varied with respect to the slope of the terrain. The slope adaptive

filter did not remove steep slopes in the terrain, however, it caused a small increase in the number of valid terrain points incorrectly rejected and an increase in the number of filter parameters (Sithole, 2001).

A method to filter ground points based on Triangular Irregular Networks (TINs) was proposed in Axelsson (2000). This technique began with a sparse TIN, created from seed points that were very likely ground points. The TIN grew denser by iteratively adapting to the data points from below. New points were added only if they met certain threshold parameters. The parameters, mainly distances to the facet planes and angles to the nodes, were derived from the data and calculated for each iteration (Axelsson, 2000). This algorithm was effective in dense city areas, since it was developed to handle surfaces with discontinuities. The adaptive TIN model method has been commercially implemented in the Terrasolid software package as part of their proprietary ground point classification routine (TerraScan, 2011).

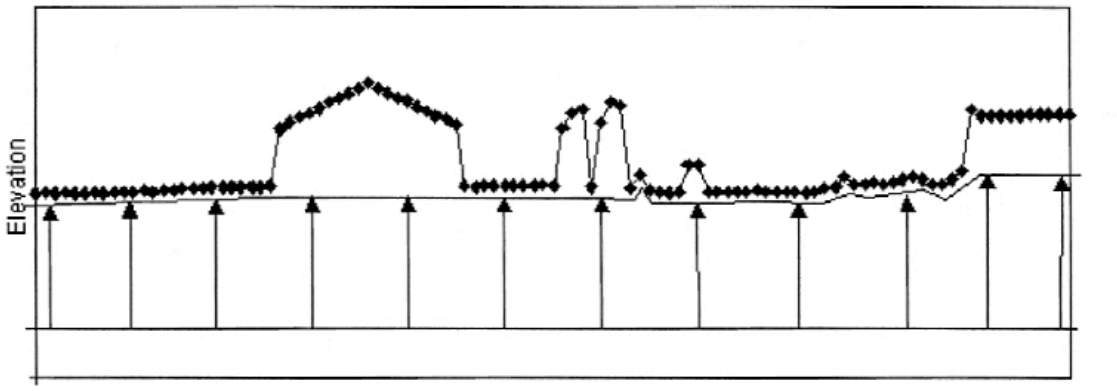


Figure 8. One-dimensional view of TIN ground surface adapting to LiDAR points. Note how well the surface is approximated from below, despite the intermittent gaps caused by buildings (Axelsson, 2000).

The second strategy for DTM extraction involves fitting functions to the LiDAR data and assigning weights to the points based on their residual values. Kraus and Pfeifer

filtered out trees using an iterative, robust interpolation process (Kraus and Pfeifer, 1998). After computing a rough approximation of the surface, the residuals or distances from the surface to each point were calculated. A weight function was used to assign each point a weight corresponding to its residual value. Larger weights were assigned to points that fell below the surface and had negative residuals, as they were likely ground points. The surface was then recomputed taking the weights into consideration, using linear prediction as the statistical interpolation method (Pfeifer *et al.*, 2001). A point with a larger weight "attracted" the surface, while small-weighted points had little effect. The process was iterated, reducing the contribution from points above the surface, so that the estimation of the ground surface approached the lowest data points (Forlani *et al.*, 2006).

The method was effective in urban wooded areas, but it relied on a thorough mixture of terrain and non-terrain points. As a result, the algorithm worked poorly in large areas without terrain points, which is generally the case in an urban environment (Rottensteiner and Briese, 2002). To address this shortcoming, the robust interpolation technique was applied in a hierarchic way using data pyramids by Pfeifer *et al.* (2001). Other techniques have been used to model the ground by fitting 3D data. Bicubic spline functions have been applied through a least squares approach by Brovelli *et al.* (2002), while active shape models were used to estimate the ground surface by Elmqvist (2002).

3.5 Building Detection from LiDAR Point Cloud

The application of LiDAR data to detect and quantify building damage is part of a relatively new field of research and relies heavily on the ability to accurately extract building features from the data set. The majority of work to date has been concerned

with building detection, extraction, and reconstruction, with scientists extracting 3D building models from airborne LiDAR data since the mid-1990s (Forlani *et al.*, 2006). In this section, several techniques for classifying aerial LiDAR data into features, specifically buildings, will be examined.

Many algorithms for building extraction that have been proposed in the literature require a normalized surface model as input. The normalized DSM is computed by subtracting the DTM, which is typically derived using one of the methods discussed in Section 3.4, from the DSM (Rottensteiner and Briese, 2002). Once the ground is subtracted from such a height-above-sea-level raster, the normalized surface model, or height model as it will be referred to in this research, contains primarily building and vegetation points. Though height thresholding can often produce an initial building mask, distinguishing between building and vegetation points remains challenging. In Brunn and Weidner (1997), a Bayesian Network classification scheme was proposed to detect buildings using three features: the height information from the normalized DSM, step edge magnitudes, and surface normal variations. This approach overcame the use of fixed thresholds, and thus produced superior classification results to binary classification (Brunn and Weidner, 1997).

A similar technique was proposed in Rottensteiner and Briese (2002), in which building detection from LiDAR points was based on Kraus and Pfeifer's robust interpolation method for DTM generation. An initial building mask was created by thresholding the difference in heights between the DSM and the DTM. As expected, some vegetation points remained and not all of the buildings were correctly segmented. A morphological opening filter was then applied to the initial building mask, with a

small, square structuring element to separate regions connected by a thin line of pixels. "Connected component analysis" was used to identify the initial set of building regions, before regions smaller than a minimum area and at the border of the DSM were removed. Some of the remaining regions may have been areas of vegetation and were discarded by evaluating terrain roughness criteria, derived from the second derivatives of the DSM. This method was shown to be effective for building extraction in densely built-up areas (Rottensteiner and Briese, 2002).

Elberink and Maas (2000) presented a technique to segment raw LiDAR data in an unsupervised classification using height texture measures. Height, variation of height in local windows, and metrics such as homogeneity and contrast were used to discriminate between buildings and vegetation. The method made the assumption that buildings have a regular, smooth pattern with small variations in height, while trees are irregular and have large height variations. Houses, sheds, and trees were classified with accuracies of 90%, 90%, and 97%, respectively. When the house and shed classes were combined into a building class, an accuracy of 98% was obtained (Elberink and Maas, 2000).

A similar technique was proposed by Charaniya *et al.* (2004), where the height model was classified into roads, grass, buildings, and trees using a supervised parametric classification algorithm. In this case, five features were used for data classification: normalized height, local height variation, multiple returns, luminance obtained from gray-scale imagery, and intensity. The results showed that height was an important classifier for terrain, and that height variation was useful in classifying high vegetation areas (Charaniya *et al.*, 2004).

One of the original techniques for extracting 3D building models using LiDAR data was presented in Haala *et al.* (1998). In this work, a planar segmentation algorithm was used to detect four basic building primitives in the DSM. The segmentation, which was based on the directions of surface normals, was supported by ground plan information. This provided additional, reliable knowledge on the relations between roof planes (Haala *et al.*, 1998). However, this implementation was limited to four standard building primitives and the availability of building ground plans.

It quickly became apparent when studying the literature that building reconstruction techniques are closely associated with building detection. Many building reconstruction approaches, used to construct 3D building models, are based on the automatic detection of planes. Though a building-class point cloud is usually required for input, the techniques to detect 3D roof planes can be applied at a higher level to aid in building segmentation. Successful detection of roof planes can also be used to find building damage, as seen later in this research. Three main methods for automatically detecting 3D building roof planes can be found in the literature, namely region growing, Hough-transform, and Random Sample Consensus (RANSAC).

Region growing techniques detect planes from rasterized height data. The starting point for each surface segment is a seed region, where all points belonging to the region lie approximately in a plane. The best-fit plane is determined by least squares adjustments. Adjacent pixels are then consecutively added to the segment if their distance to the plane is below some threshold. When pixels can no longer be added to a segment, additional seed regions are selected and expanded until no more seed regions can be found. Unlike other plane detection techniques, region growing results in a set of

unsegmented pixels in addition to the plane surface regions. This prevents segmentation errors from occuring when planes are fit to points that do not lie on the same plane (Vosselman, 2009).

The 2D Hough transform is used in image processing to detect geometric primitives such as lines, circles, and ellipses. It works by representing a set of points in an image, defined initially in Euclidean space, in a parameter space (Tarsha-Kurdi *et al.*, 2007). A point *(x,y)* in an image, for example, defines a line $y = ax + b$ in the parameter space, where the parameters a and b form the axes. If several points in an image lie on a straight line, the lines of the points in the parameter space will intersect, and the point of intersection represents the parameters of the line in the image (Vosselman and Dijkman, 2001).

This principle has been extended to 3D space, where each point *(x,y,z)* in a point cloud defines a plane $z = s_x x + s_y y + d$ in the 3D parameter space spanned by s_x, s_y, and d. The slopes in the x- and y-direction are represented by s_x and s_y, while d is the vertical distance from the plane to the origin. The intersection point of planes in the parameter space corresponds to the slopes and distance of the planar face in the point data (Vosselman and Dijkman, 2001). The 3D Hough transform looks for point sets that statistically represent the best planes, meaning the plane containing the maximum number of points. This method is susceptible to detecting a set of points which represent several roof planes, since context information from the building point cloud is not taken into account (Tarsha-Kurdi *et al.*, 2007). In addition, the algorithm requires discrete intervals on the s_x, s_y, and d axes. If small step sizes are chosen, the quality of the detected plane is better, but the processing time and memory requirements increase. Calculating the

matrix that represents the point cloud in parameter space can be very time and memory intensive. It is also very difficult to determine the parameters automatically, since they are related to the characteristics of the building roof planes and the point cloud (Tarsha-Kurdi *et al.*, 2007). A benefit of the Hough transform is that it does not require surface normal vectors to be calculated, which can be very noisy in the case of high point density datasets (Vosselman and Dijkman, 2001).

Finally, RANdom SAmple Consensus (RANSAC) represents another algorithm used to detect planar faces in irregularly distributed point clouds. This technique uses an iterative approach to search for the best plane. Three points are selected randomly, and the parameters of the corresponding plane are calculated. All the points from the point cloud belonging to the calculated plane are detected, using a given tolerance threshold of distance t. The process is then repeated N times, comparing the results from each iteration to the previous saved results, until the best plane is found (Rehor *et al.*, 2008). A comparison of the 3D Hough transform and RANSAC was performed in Tarsha-Kurdi *et al.* (2007). The authors concluded that RANSAC not only provided results in a shorter time, but also had a higher percentage of successfully detected planes. Like the Hough transform, RANSAC is based on pure mathematics without the building cloud's context, so a set of points may be detected that represent several roof planes, or which belongs to several planes.

As the diversity of remote sensing technologies has increased over time, the trend has been to fuse LiDAR data with that of imaging modalities. The detection of building edges can be very difficult using only LiDAR data, which often leads to problems detecting roof planes. Vegetation points near roof edges, or points on nearby roofs of

similar height are often incorrectly segmented. Better segmentation of the point cloud and more accurate roof plane detection can be achieved with a priori knowledge of the data and their context. The presence of ground plans, for example, constrains the search space and allows for the handling of complex buildings. Ground plans, which give insight to building footprints, were used by Haala *et al.* (1998), Vosselman and Dijkman (2001), and Alexander *et al.* (2009). Orthorectified aerial and satellite imagery can also be fused with LiDAR data to aid in building detection. Imagery is often used to refine the initial LiDAR segmentation by detecting sharp building edges. Multi-band images can be used to spectrally classify objects, using techniques such as Normalized Difference Vegetation Index (NDVI) and Normalized Difference Water Index (NDWI).

An automatic building extraction method that fused IKONOS imagery with LiDAR data was proposed in Sohn and Dowman (2007). The algorithm started with an initial building set, achieved by applying a threshold to the normalized DSM and selecting all the points above a certain height. NDVI was then used to distinguish the buildings from other features. NDVI takes advantage of the fact that the spectral reflectance of vegetation abrubtly increases from the red to the near-infrared spectral regions. The formula, which uses differences and ratios to reduce illumination, calibration, and atmospheric correction effects is expressed as

$$NDVI = \frac{DC_{IR} - DC_R}{DC_{IR} + DC_R}, \tag{2}$$

where DC_{IR} and DC_R represent the digital count values in the IR and red spectral bands, respectively (Schott, 2007). A global threshold was applied to the NDVI map to divide

the features into building and tree classes (Sohn and Dowman, 2007). Similar fusion methods were presented in Chen *et al.* (2009) and Vu *et al.* (2009).

3.6 Recognizing Damaged Buildings in LiDAR Data

The benefits of using LiDAR for detection and classification of building damage are just being recognized, and as a result literature on past research remain scarce. A potential reason for the lack of methods is the difficulty of obtaining real LiDAR data acquired after a disaster, which can be used to develop and test techniques. To date, much of the work investigating building damage following a disaster has been performed using change detection. These approaches require both pre- and post-event LiDAR datasets of the affected area. In Vögtle and Steinle (2004) and Vu *et al.* (2004), change detection was performed on two building point clouds collected at different times. These methods generally classified buildings into several different categories: not-altered, added-on, reduced, new, and demolished.

Additional approaches seek to further classify building damage by comparing planar surfaces extracted from LiDAR data with roof planes of reference building models. Rehor (2007) divided each building into several segments and assigned each segment to one of ten different damage types. A catalog of the damage types of buildings after earthquakes, shown in Figure 9, was developed by Schweier and Markus (2004). The segments were assigned to classes based on the following features, which were described in the catalog for each damage type: volume reduction, height reduction, change of inclination, and size. All of the features, except size, required pre-event data for comparison.

3.7 Chapter Summary

A review of the literature on DTM extraction, building segmentation, and damage detection using LiDAR data highlights the complexity of these tasks. They each can be accomplished any number of ways and there is no clear consensus on leading approaches. What is evident, however, is that these topics rely heavily on point classification. Whether classifying points as ground to model the terrain surface, or classifying non-ground points as building points to detect man-made structures, accurate point classification is the underlying theme. The line between building segmentation and damage detection is also blurred. For example, some of the same features that are used to detect buildings, such as texture and dominant roof planes, can also be used to assess damage.

Many of the algorithms proposed in the literature require the use of supervised training sets, data fusion with other remote sensing modalities, or are very time intensive. The major goal of this research is to develop a tool that can produce damage maps rapidly, i.e., a turnaround time of less than 24 hours, and with as little human interaction as possible. The algorithms also have to work on a variable dataset, as the region surrounding Port-au-Prince is representative of a variety of building shapes, sizes, materials, and terrain properties. Armed with the background knowledge, and keeping the intended purpose and scope of the research in mind, the specific experimental methods are determined next.

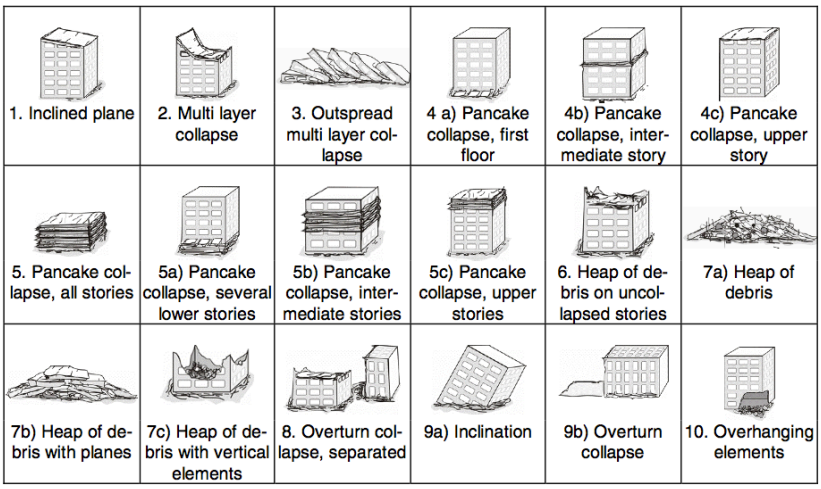


Figure 9. Catalog of different damage types of buildings occurring after earthquakes (Schweier and Markus, 2004).

4 Methodology

4.1 Chapter Overview

The objective of this research was to develop an end-to-end operational tool that will ingest a LiDAR point cloud of a post-disaster scene, and output a geo-referenced building damage map showing damaged and collapsed structures. The tool was developed and tested using post-earthquake LiDAR data of Haiti. The workflow can be broken down into three distinct tasks. First, the effect of the terrain was removed from the raw point cloud to derive a normalized surface model. Second, the building points were separated from the rest of the point cloud using a combination of point classification techniques found in the literature. Lastly, damage was detected by measuring the deviation between building roof points and dominant planes. This workflow is illustrated in Figure 10, which should be used as a reference throughout this chapter. The algorithms were coded in Matlab R2009a and implemented in a Matlab graphical user interface (GUI) that resembled the final tool and allowed for user interaction. The main menu of the software program is shown in Figure 11.

4.2 World Bank/ImageCat Inc./RIT Haiti Earthquake Dataset

In the days immediately following the Haiti earthquake, the World Bank contracted with ImageCat Inc. and RIT's Chester F. Carlson Center for Imaging Science to collect high spatial resolution, airborne, multi-spectral imagery and LiDAR data over the affected area. The dataset was collected January 21-27, 2010, using RIT's WASP camera system and a Leica ALS60 LiDAR instrument operated by Kucera International (Messinger *et al.*, 2010; Faulring *et al.*, 2011). During the seven-day campaign, data

collection occurred over regions in and around Port-au-Prince, Léogâne, Petit- and Grand Goâve, Fermate, Jacmel, and the western and eastern fault lines. The data collection area is shown in Figure 12, overlaid on a map of Haiti.

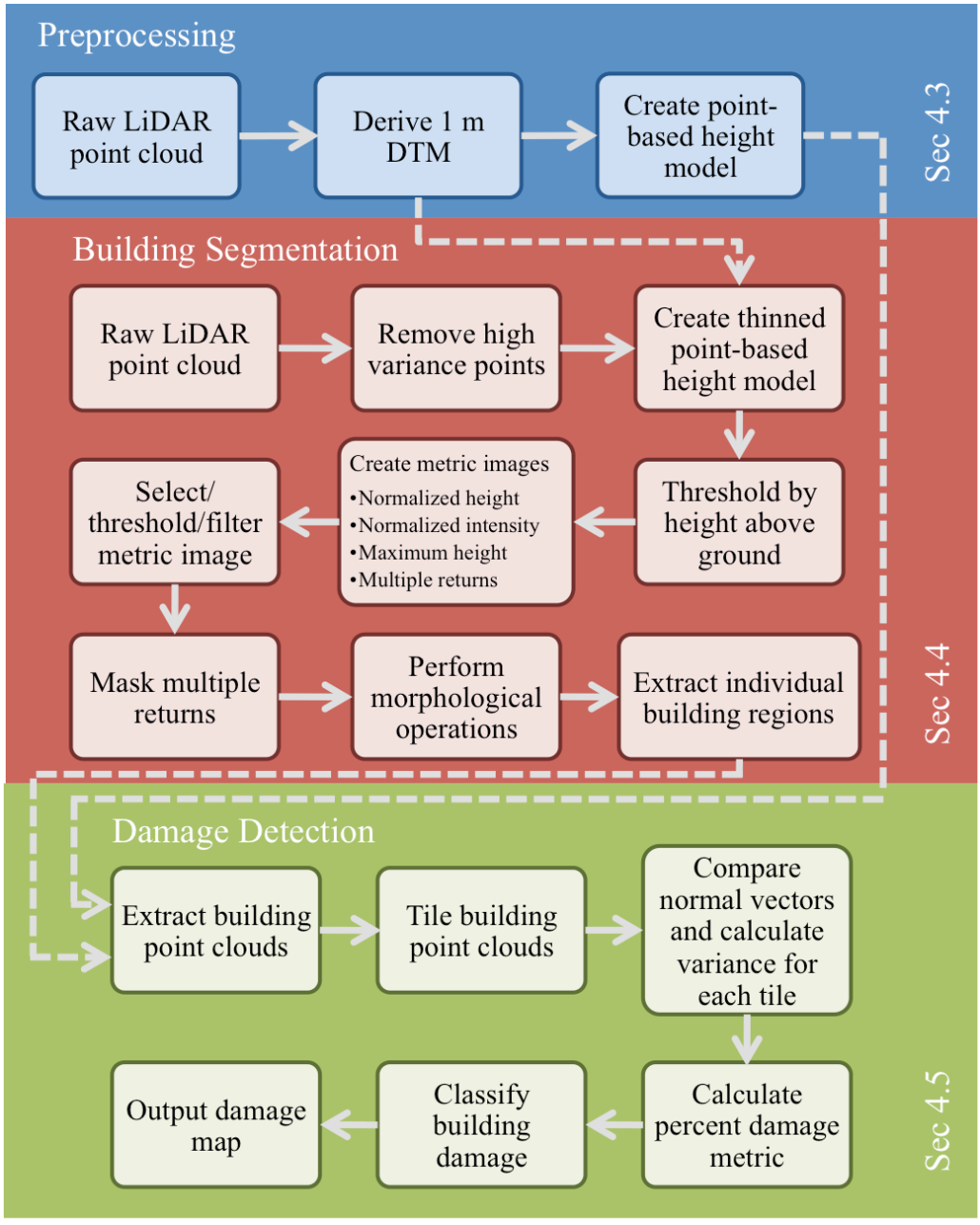


Figure 10. Comprehensive flowchart showing the workflow broken down into three distinct tasks: Preprocessing, building segmentation, and damage detection. Dotted lines represent links between tasks, as inputs needed for certain processes come from earlier stages of the workflow.

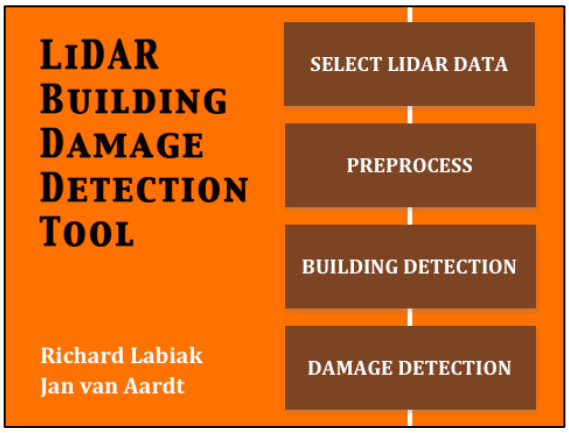


Figure 11. Main menu of the Matlab-based LiDAR Building Damage Detection Tool.

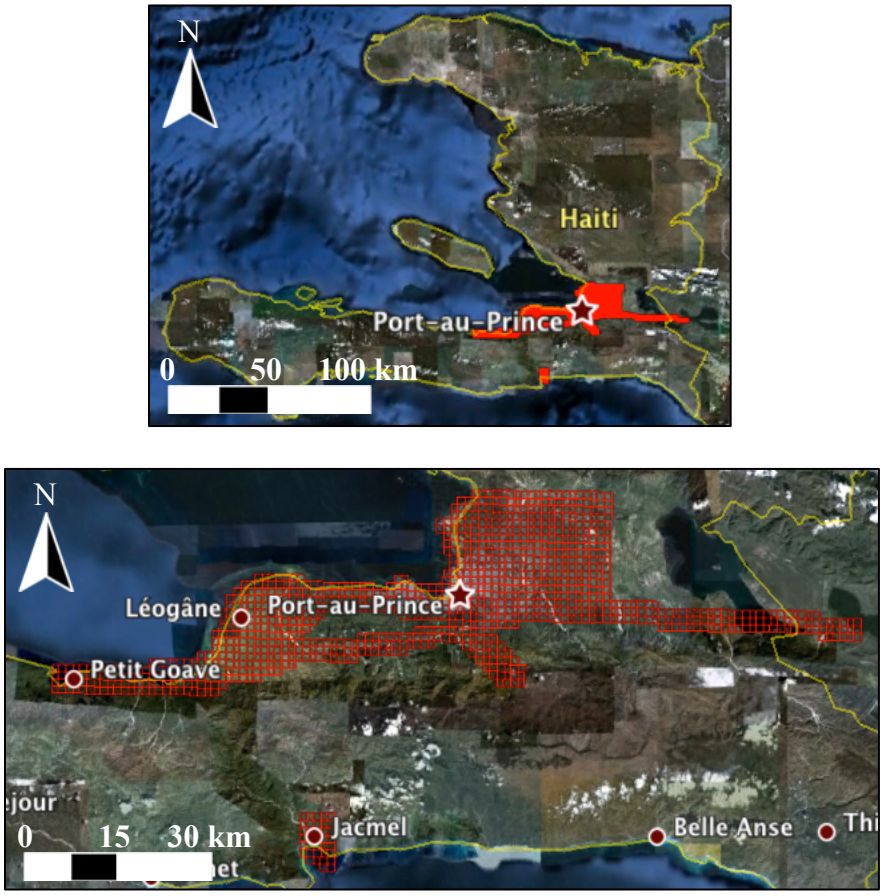


Figure 12. *Above:* Map of Haiti showing the data collection region in red. *Below:* Zoom-in view of the area affected by the earthquake. Each 1 km x 1 km tile outlined in red represents an individual LAS file containing on average 3 to 5 million LiDAR points (Google Earth).

Since the WASP imagery collection was the priority for the World Bank, who sponsored the campaign, the LiDAR data collection was governed by the image collection specifications and flight design. The LiDAR dataset was collected using a Leica ALS60 airborne laser scanner, emitting light pulses at a wavelength of 1064 nm at an altitude of approximately 820 m. The instrument was operated at pulse rates of up to 150 kHz, which resulted in a point cloud density of roughly 2-5 points/m^2, given the prevailing imagery collection parameters. The vertical placement accuracy was approximately 0.11 m. The dataset was initially processed and delivered by Kucera International in LAS file format, i.e., an open, binary format that contains LiDAR point data records. Each LAS file covered a 1 km x 1 km area on the ground. The LAS tiles are shown outlined in red in Figure 12. Each point record in the Haiti LAS files contains the *X, Y, Z* location for the return, return intensity, return number, number of returns for that pulse, and the vendor-supplied ground vs. non-ground point classification. The latter was based on the proprietary slope-radius algorithm in Terrasolid LiDAR workflow (TerraScan, 2011). The LiDAR dataset covers an area of 838 km^2, consists of 2,853,027,995 individual returns, and requires 75 GB of hard disk space (OpenTopography).

The WASP sensor, positioned on the same platform as the Leica ALS60, was used for collecting visible (RGB), shortwave-infrared, midwave-infrared, and longwave-infrared imagery over Haiti. The imagery was orthorectified; its co-location with the LiDAR data makes it valuable for potential fusion applications and validation during LiDAR algorithm development and testing. Figure 13 shows a 0.15 m spatial resolution

WASP image and the corresponding LiDAR point cloud of the city of Darbonne, located southeast of Léogâne.

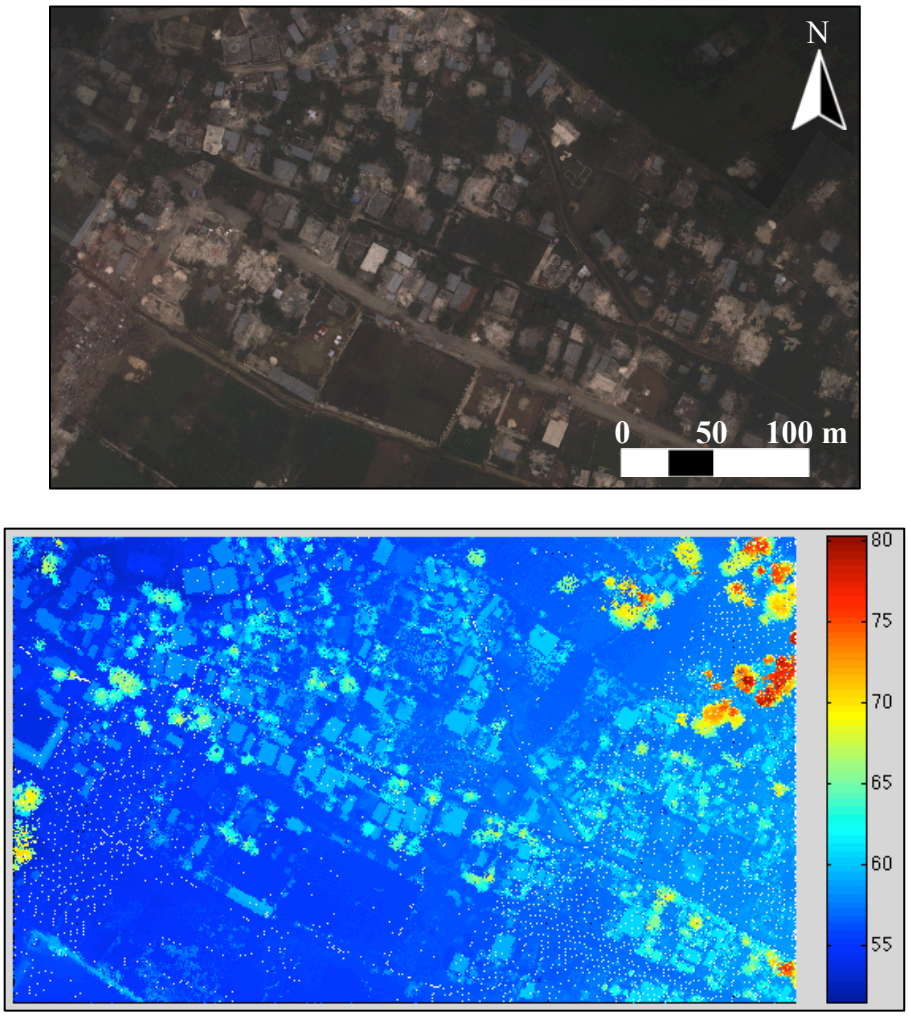


Figure 13. Scene captured over the city of Darbonne on January 25, 2010. *Above:* WASP image with a spatial resolution of 0.15 m. *Below:* Corresponding raw LiDAR point cloud colored by elevation. Point heights in the scene range from 51 m to 80 m, before the effect of terrain is considered.

4.3 LiDAR Point Cloud Preprocessing

The majority of the building identification and damage detection techniques described in the literature are based on a terrain normalized DSM as a starting point, i.e., referencing the DSM or LiDAR point heights relative to the LiDAR-derived DTM. Eliminating the effect of the terrain, which effectively references the point cloud above

ground, can quickly produce an initial building mask. By setting a global threshold at 1 m, for instance, all the points with elevations below can be removed leaving primarily building and vegetation points. In addition, prior to any substantial processing, gross outlier points should be eliminated. Such points can be caused by bird hits, or more likely noisy data. Though these gross outlier points make up a very small percentage of the total points in the point cloud, they can induce errors if not removed. During the early stages of this research, gross outlier removal was accomplished autonomously by removing points higher than three standard deviations above and lower than three standard deviations below the mean height of the entire point cloud. Unfortunately, this often removed high vegetation and building points as well, since the point clouds of Haiti consist of many low and ground points, with an associated small standard deviation. In the current workflow, the first step of the proposed building detection algorithm involves computing the variance of the raw point cloud and removing high variance points, so the gross outlier points are removed early in the process.

The main goal of the preprocessing applied here is to reference the LiDAR point cloud above ground, thereby removing the effect of varying topography. As mentioned in Section 3.5, the normalized DSM is created by subtracting the DTM from the DSM. The bulk of the work during this step involves the creation of a DTM that accurately represents the ground surface topography. Ground points must be separated from non-ground points, which is often challenging. Several DTM extraction and point classification approaches are described in Section 3.4.

The Haiti LiDAR dataset includes a vendor-supplied ground point classification, which is leveraged by this research. Kucera International classified each point using

Terrasolid, a commercial software program. The Haiti LiDAR returns were classified into three standard point classes: unclassified, ground, and low points (or noise). Several classification routines were implemented in Terrasolid to perform the classification. The first routine, which filtered isolated points, classified returns as low points if they did not have a given number of neighbor points within a 3D search radius. Second, the ground classification routine classified bare earth points by iteratively building a triangulated surface model. This routine, based on Axelsson's adaptive TIN model (Axelsson, 2000), began with a selection of a few local minimum points that were confident hits on the ground. The initial point selection was controlled by a user-defined maximum building size parameter, so the software could determine a minimum area where there will be a least one ground hit. The routine then triangulated a surface between the selected minimum points. The ground model was then slowly adapted upwards and refined by iteratively adding additional points. Iteration angle and iteration distance, illustrated in Figure 14, were two parameters that needed to be met for a point to be included in the model. These user-defined parameters were optimized for the terrain. Typical values for iteration angles range from 4° to 10°, where 4° is chosen for flat terrain and 10° is selected for hilly terrain (TerraScan, 2011).

The third Terrasolid routine located points that were below the true ground surface and assigned them to the low points or noise class. The algorithm worked by fitting a plane equation to the 25 closest ground points surrounding the point of interest. The standard deviation of the elevation differences from the neighboring points to the plane was computed, and if the point of interest was less than the computed value times a user-defined constant, it was classified as a noise point. Lastly, a low points routine was

run that classified points that were clearly lower than other points in the vicinity (TerraScan, 2011).

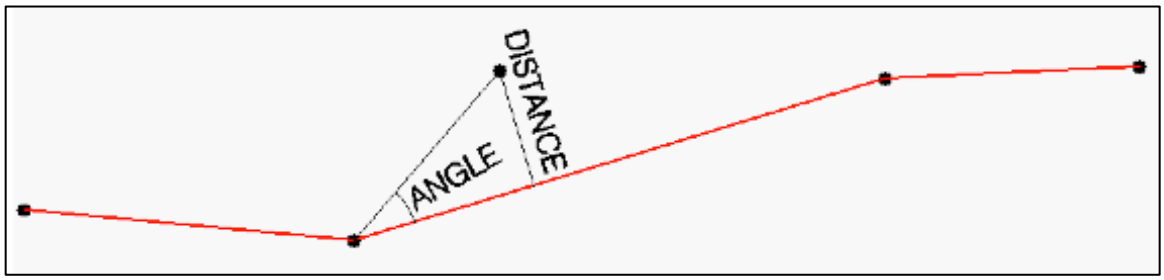


Figure 14. Angle and distance parameters must be met every iteration for a point to be added to the ground surface model. The iteration angle is defined at the maximum angle between a point, its projection on the triangle surface, and the closest triangle vertex. The iteration distance is defined as the maximum distance from a point to the triangle surface (TerraScan, 2011).

Kucera International provided details regarding the classification when they delivered the LAS files. The user was warned about the quality of the classification, mentioning that the turnaround time due to the unfolding disaster allowed for very little manual editing. According to the vendor, "Urban areas were of sufficient uniqueness and complexity to require a catch-all approach due to lack of exterior background information on the unique properties in the scene." In some urban areas very few points were classified as ground, given the building density in these areas.

To create the DTM, the irregularly spaced ground points were interpolated onto a 1 m x 1 m grid. The 1 m grid resolution was chosen since it was dense enough to capture elevation changes in rugged terrain, yet still could be supported by the low point density of the ground hits (approximately 0.2-0.6 ground hits/m^2). Though there are several methods that can be used to interpolate scattered data, Delaunay triangulation using natural neighbor interpolation was selected to construct the DTM. The choice was based on a comparison between nearest neighbor interpolation, natural neighbor interpolation, and inverse distance weighting. Nearest neighbor interpolation is simple to implement,

and approximates the value of an unsampled point using the value of the nearest point. Nearest neighbor does not produce as smooth of a surface as natural neighbor interpolation, which determines point values through a weighted average of the interpolating neighbors. The natural neighbors are selected by finding the Voronoi polygons, or tessellations, around the sampled points that would intersect with the Voronoi region created around the interpolation point. The weighting factor is based on the percentage of overlap. Like the natural neighbor approach, inverse distance weighting (IDW) is also based on weights; however, weights are assigned based on distance from the interpolation point. In this case, the neighborhood can be defined using a given number of neighbors, or using a circle with a given radius around the unsampled point (ESRI, 2010).

The three interpolation techniques were evaluated on a scene surrounding Haiti's National Palace, consisting of 117,221 LiDAR ground points. It took 3.1 s and 5.4 s for the nearest neighbor and natural neighbor algorithms to run, respectively, while the execution time for IDW using a neighborhood of 30 points was just shy of two hours. The execution times were calculated using a 2.66 GHz MacBook Pro laptop. Though there was no truth DTM of the scene available that could be used to assess accuracy, the 1 m resolution surfaces were compared against each other to quantify the differences. A root-mean-square (RMS) approach on the difference between grids was used to compute an overall metric of similarity. The average difference between natural neighbor and nearest neighbor was found to be 11.4 cm, between natural neighbor and IDW was 9.0 cm, and 11.5 cm between nearest neighbor and IDW. These differences, which themselves only differed 2.5 cm, were similar enough to suggest that one approach was

not performing significantly better or worse than another. A visual inspection of plots of the three surfaces confirmed this, as no distinct differences were noticed. As a result, the computationally intensive IDW was not selected. Despite having a slightly longer execution time, on the order of a couple seconds, natural neighbor interpolation was chosen over nearest neighbor due to the fact that it approximated smoother surfaces, which closer resemble terrain.

Figure 15 shows the DTM of the region surrounding Haiti's National Palace, from both an oblique and nadir view. With the DTM created, the effect of the terrain was effectively removed from the point cloud. This resulted in the normalized DSM, or "height model" as it will be referred to from this point forward. Two options were considered when creating the height model. The first was to subset the full point cloud to only first return points, and interpolate the points onto a 1 m grid. First returns represent the first surface with which the light pulse interacts, which implies that this interpolated surface is the top-most feature surface. The DTM was simply subtracted from the surface model, resulting in a rasterized set of points referenced to the ground. The second option was a point-based approach, which used the "unclassified" points, or those determined to be non-ground. For each non-ground point, the *X,Y* location in the DTM grid closest to the *X,Y* position of the point was found. The height of the terrain at that location was then subtracted from the height of each point. This point-based approach was used to create the height model, since it resulted in a denser point cloud, which was later needed to detect buildings and quantify building damage. Also, unlike the raster approach it did not use interpolation, which could lead to spurious data points. Figure 16 illustrates the process of creating a height model for the scene captured over Darbonne. The height

models of Darbonne created using the two different methods are shown in Figure 17. Of the 370,378 points in the raw point cloud, the raster-based height model consisted of just 106,848, arranged in a regular, 1 m grid. This is far less dense than the point-based height model, which consisted of 341,784 non-ground points.

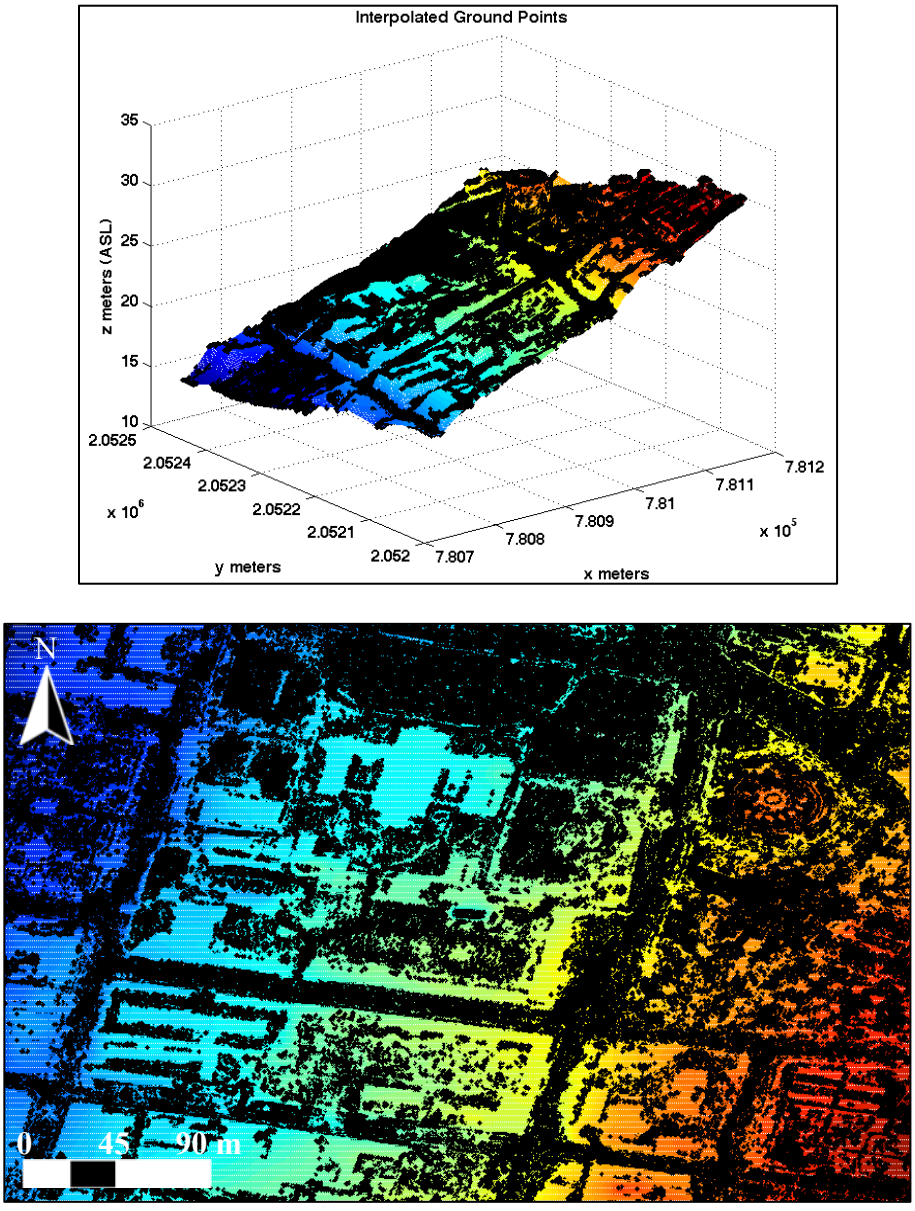


Figure 15. One-meter resolution Digital Terrain Model (DTM) of the region surrounding Haiti's National Palace. The colored mesh represents the ground surface approximation, and the black points are the vendor-classified ground points. *Above:* Oblique view with a stretched z-axis to show elevation change. *Below:* Nadir view.

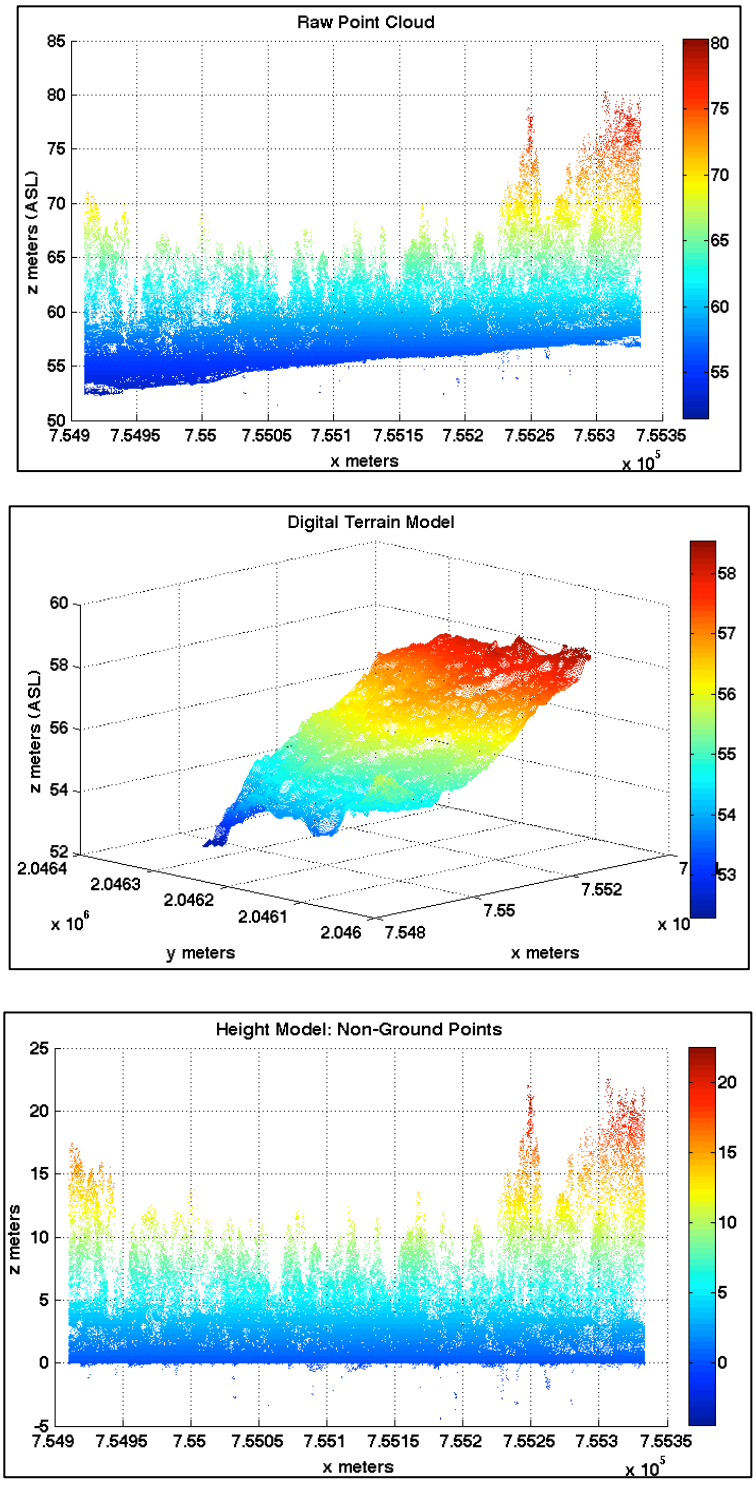


Figure 16. The effect of the terrain is eliminated from the point cloud of the city of Darbonne. The points are colored by elevation according to the colormap for each plot. *Top:* Raw point cloud prior to DTM extraction. *Middle:* DTM approximated using natural neighbor interpolation of the ground points. *Bottom:* Height model created by subtracting the height of the terrain from each non-ground point.

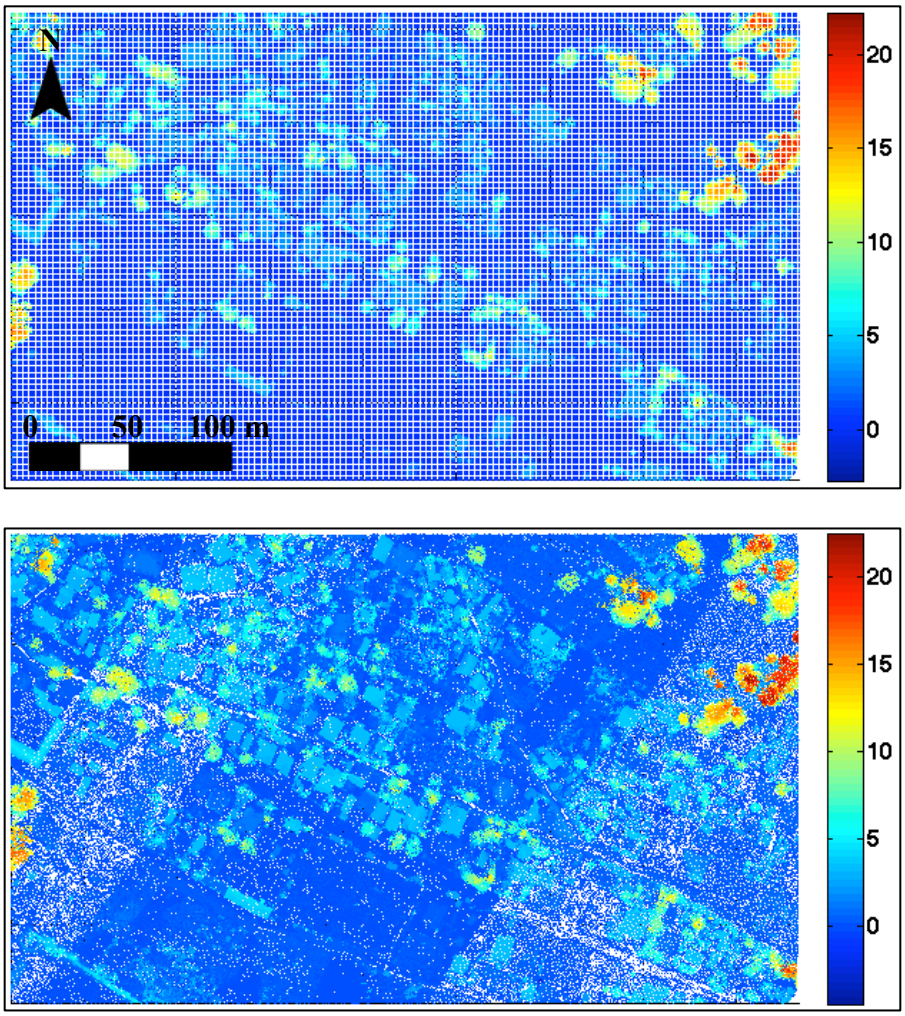


Figure 17. Height models of Darbonne created using two different techniques. *Above:* First return points are interpolated and subtracted from DTM to create a *raster* height model. *Below:* Point-based height model created by subtracting the corresponding DTM height from non-ground points.

4.4 Building Segmentation

Originally, an automated method for detecting buildings from a LiDAR point cloud was developed for the scene around Haiti's National Palace. The process, which involved global height-above-ground thresholds, intensity and multiple return information, and morphological filtering, was optimized to extract individual building regions and produce a building mask. The algorithm was based on a normalized intensity metric, which was calculated by dividing the average intensity of the points in a 1 m x 1

m cell by the maximum intensity value in the cell. The assumption was that local areas with relatively uniform intensity would result in cell values close to one, which could be useful when distinguishing buildings from vegetation in a scene. The method failed, however, when it was extended to other study areas. The buildings in other scenes did not all have normalized intensity values near one, so they blended with the vegetation. This made extraction using a global threshold difficult.

The failure can be attributed to the large variation in building size and building materials found in Haiti. According to the Haitian Ministry of Statistics and Informatics, most ordinary, one-story houses have roofs made of sheet metal, while most multi-story houses and apartments have roofs made of concrete (Eberhard *et al.*, 2010). Table 1 shows the percentage of housing units according to roof type. There were two types of buildings that were routinely encountered during this research. Both are shown in Figure 18: the first was shanty housing constructed using a mixture of wood and corrugated metal, while the second was concrete and masonry buildings with concrete slab floors and roofs.

Table 1. Percentage distribution of housing units according to roof type. The Kay atè, Taudis, and the Ajoupas are housing for the poor and extremely poor (Eberhard *et al.*, 2010).

Type of Housing	Type of Roof				Total
	Concrete	Sheet Metal	Straw	Thatch/Palm leaves/others	
Kay atè (combined roof and walls)	-	-	47.3	52.7	100.0
Taudis/Ajoupas	-	44.7	40.6	14.7	100.0
Ordinary One-Story House	10.5	81.9	5.8	1.8	100.0
Ordinary Multistory House/Apartment	70.9	20.4	-	8.7	100.0
Others	7.1	69.5	13.2	10.2	100.0
All	13.9	64.9	13.2	10.2	100.0

Figure 18. Two prominent building construction types found in Haiti. *Above:* Shanty housing made of wood and a corrugated metal roof. *Below:* Residential buildings constructed of reinforced concrete columns, infill concrete walls, and concrete slab floors and roofs (Eberhard *et al.*, 2010).

The normalized intensity metric worked relatively well on the large, flat roofs that made up most of the scene surrounding the National Palace. This can be seen in Figure 19, where individual buildings could be extracted if a global threshold of approximately 0.85 - 0.9 were applied to the image. The normalized intensity metric was not as helpful in the Darbonne scene, due to the small building sizes and non-uniform intensities caused

by the roof materials. Not only were some buildings completely omitted when a threshold of 0.9 was applied, but many "noisy" pixels remained which rendered building extraction impossible.

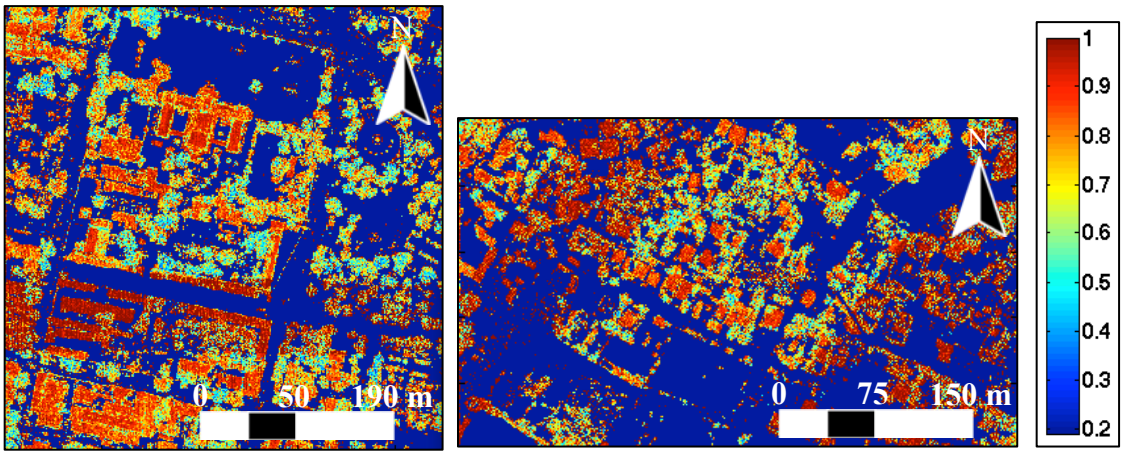


Figure 19. Normalized intensity metric used to identify building regions. Building regions are assumed to have uniform intensity, so building pixel values should be close to one. *Left:* Haiti's National Palace and surrounding area. *Right:* The Darbonne area.

It therefore was decided to move away from developing a fully automated method for building detection, and instead create a tool that provided a workflow guided by user inputs, due to the high variation of building sizes, building materials, and building conditions in Haiti. The resultant interactive tool allows users to choose operations and parameters that best fit the scene they are evaluating. In addition, the normalized intensity metric was no longer the basis of the building detection algorithm. Several techniques were combined to distinguish building regions from vegetation. The overall methodology, following the workflow of the tool, will be described in the remainder of this section using the Darbonne point cloud as an example study area.

The building detection workflow was designed to gradually identify and remove non-building points from the point cloud, so only building regions remain. The first step in this process was to remove points with large height variances, which is characteristic

of vegetation points. This was performed on the raw point cloud, since there was no need for the point heights to be referenced to ground. Height models are only as good as the ground surface approximations they are based on, making them prone to errors. As a result, it is better to operate on the raw point cloud when possible. The raw point cloud then was tiled, using a user-defined tile resolution, and the height variance of the points in each tile was calculated.

The algorithm removed the tiles with the highest variances, by initially finding the variance values that were greater than three standard deviations above the mean. These "outlier" variances were then removed, and the mean and standard deviation of the variances were recalculated. After investigation, it was determined that tiles with height variances greater than two standard deviations above the mean generally contained vegetation points and should be removed. All the points within the selected tiles were then removed from the raw point cloud. The resulting point cloud, with high variance points removed, is shown in Figure 20. A tile resolution of 2 m x 2 m was chosen for the point cloud of Darbonne, corresponding to approximately 14 returns per tile. It should be noted that the tile size should be less than the size of the smallest building in the scene, but also be large enough to contain a statistically useful number of points.

From Figure 20 it is evident that removing the tiles with high height variance is a successful first step in removing vegetation regions, while keeping building regions intact. Though a lot of small trees were completely removed, the technique tended to leave some vegetation points in the middle of large tree canopies. This was attributed to the fact that in those dense areas, the height variance between points was small, compared to the large variances between points on the canopy edges and the ground. The next step

in the workflow was to remove these remaining vegetation tiles. This was accomplished by creating a binary mask image of the tiles, where removed tiles were assigned a value of zero, and all the rest of the tiles were given a value of one. A Gaussian lowpass filter of a user-defined size was then applied to the tile mask, and all the tiles with values less than one, and therefore influenced by the discarded tiles, were flagged for removal. The binary masks are shown in Figure 21. Figure 22 shows the raw point cloud of Darbonne with the additional tiles removed. In this case, a Gaussian filter size of 2 x 2 was chosen, since it performed well at removing the majority of the residual vegetation tiles, without encroaching on the building regions.

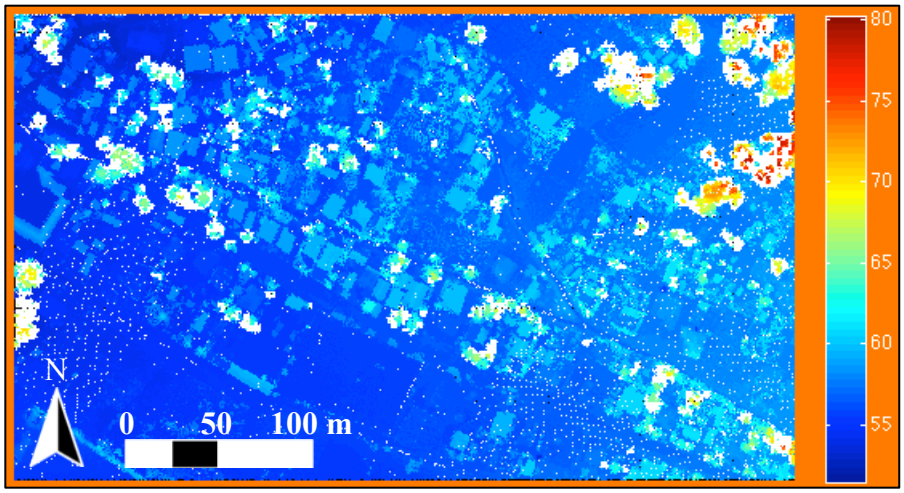


Figure 20. Vegetation points are removed from the raw point cloud by finding points with high height variance. The tile resolution used was 2 m x 2 m, resulting in approximately 14 points per tile.

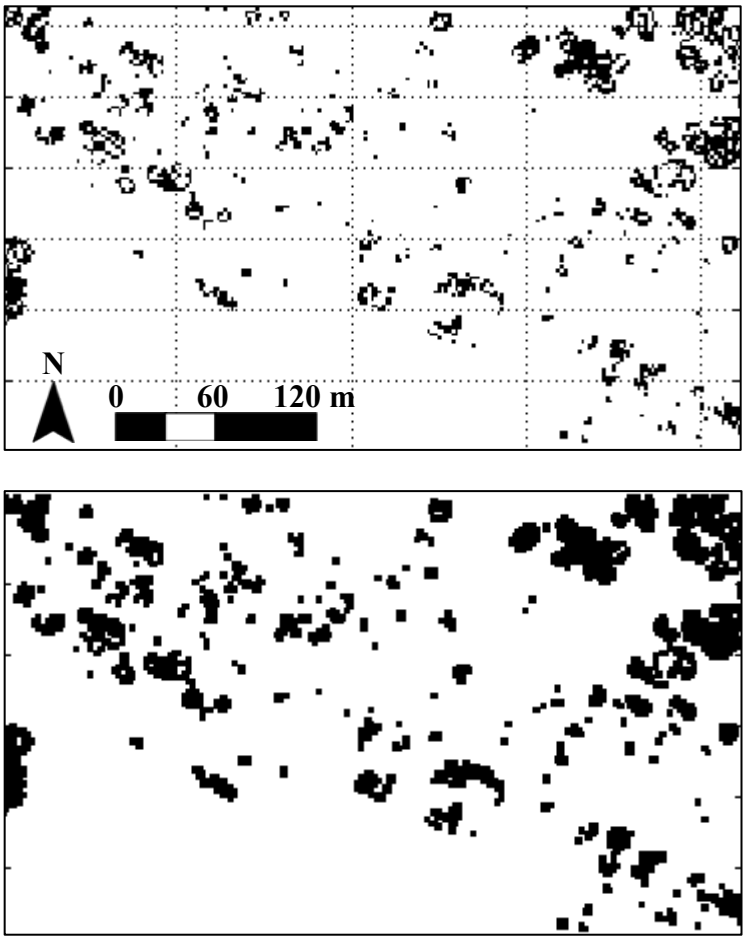


Figure 21. Binary masks with the tiles flagged for removal shown in black. *Above:* Mask created by assigning a value of one to high height variance tiles (black). *Below:* Mask after applying a Gaussian lowpass filter with a 2 x 2 kernel.

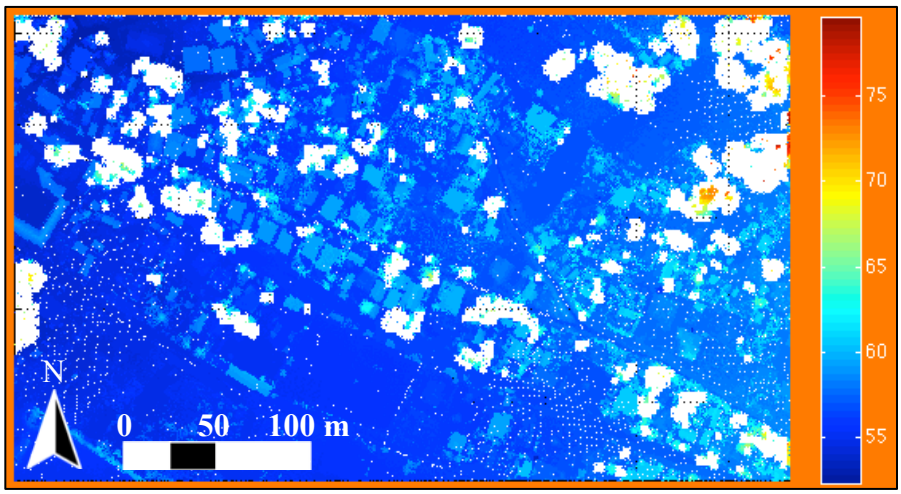


Figure 22. Raw point cloud with additional vegetation tiles removed as a result of Gaussian lowpass filtering.

Although the bulk of the vegetation points were now theoretically removed, there will always be residual points. In the next step, additional vegetation points were removed by applying a height threshold to the point cloud. The user selected the threshold, such that points above were assumed to be vegetation points, and discarded. Using the profile view of the Darbonne point cloud in Figure 23, a threshold of 62 m above sea level was chosen for point removal. The resulting point cloud, with the additional points removed, is shown in Figure 24.

The next major step in the workflow was to reference the point cloud to the ground, and create the height model. This was accomplished using the point-based approach explained in Section 4.3, by subtracting the DTM derived during the preprocessing stage. With all the point heights normalized, a global threshold was then applied to remove points below a user-defined height: the assumption was that all the buildings in the scene would be taller than the user-defined threshold. Even if a building was collapsed, the debris pile should rise above the height value chosen. All points with heights below 2 m were removed from the Darbonne height model. Once the threshold was implemented, the resulting point cloud, shown in Figure 25, contained 76% fewer points. With the majority of the ground, roads, curbs, shrubs, cars, etc. removed, it was safe to assume that the remaining point cloud consisted of either building or vegetation points.

The user could then choose from a variety of metrics to remove non-building points and segment the point cloud into individual building regions. The metrics, which included normalized height, normalized intensity, maximum height, and number of multiple returns, were calculated by gridding, or rasterizing, the 3D point cloud.

Depending on the metric of interest, different operations were performed on the points in each grid cell, producing a set of raster images. To calculate the normalized height of the points in each grid cell, the mean height value was divided by the maximum height value. A similar approach was used to calculate normalized intensity, where point intensity values were used instead of point heights. These two techniques were based on the idea that buildings should have uniform heights and intensities, so they should produce values closer to one than other features in the scene. The maximum height metric was implemented by finding the maximum height of the points in each cell. This was a useful metric for scenes in which features have defined, unique heights, such as city buildings and skyscrapers that dwarf trees. This was not the case in Haiti, where the trees and buildings could be the same height, making height-based segmentation challenging. The multiple returns image was calculated by simply counting the number of multiple returns in each grid cell. This metric provided insight specifically into the location of vegetation in the scene. Images resulting from the four operations are shown in Figure 26. A 1 m x 1 m grid size was chosen, based on the point density of the data. Smaller grid cell sizes produced rasters with empty cells, and took longer to process, while larger cell sizes started to blur building edges and resulted in a loss of well-defined structure.

Once created, the images calculated from the different operations were then assessed for their utility in building segmentation. The normalized height, normalized intensity, and multiple return images were evaluated to see which would result in the most accurate set of building regions once a threshold was applied. In the case of Darbonne, the normalized height metric was used, with an initial threshold of 0.9. It is clear from Figure 26 that when pixels below 0.9 were deleted from the normalized height

image, most of the non-building "noisy" pixels were removed, while the majority of building regions were preserved. In this case, thresholding the normalized intensity image did not provide as accurate a set of building regions, since the normalized intensity metric did not perform as well on the smaller building sizes and non-uniform building materials in the scene. As expected, the maximum height metric did not help distinguish buildings from vegetation, given the prevalence of lower buildings in the Haiti scenes.

Figure 27 shows the thresholded normalized height image. A Gaussian lowpass filter was applied to remove the noise and aid in building segmentation. The resulting image, shown in Figure 28, could then be thresholded to remove the non-building pixels. All the pixels with values below 0.8 were removed from the image, and distinct building regions appeared. This initial building map can be seen in Figure 29.

The next step in the building detection workflow was to leverage the multiple return information with the intent of removing the remaining vegetation pixels. The same Gaussian lowpass filter was applied to the multiple return image shown in Figure 26. The smoothed vegetation pixels were then used to create a binary multiple return mask, which is shown in Figure 30. The mask was then overlaid on the initial building map, and any overlapping pixels were removed from the map. The resulting building map is shown in Figure 31. A flicker test reveals that a limited number of building pixels were removed from the building map after the multiple return information was incorporated, but likely not enough to make a difference in the overall building segmentation result. Though this step was not regarded as effective for this region, it did validate earlier tasks in the workflow, given its success in removing vegetation.

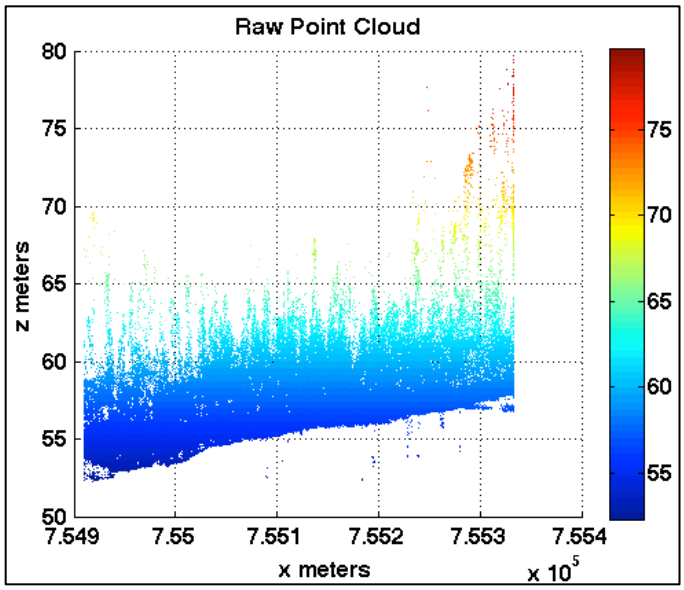


Figure 23. A profile view of the Darbonne point cloud used to determine the best height threshold.

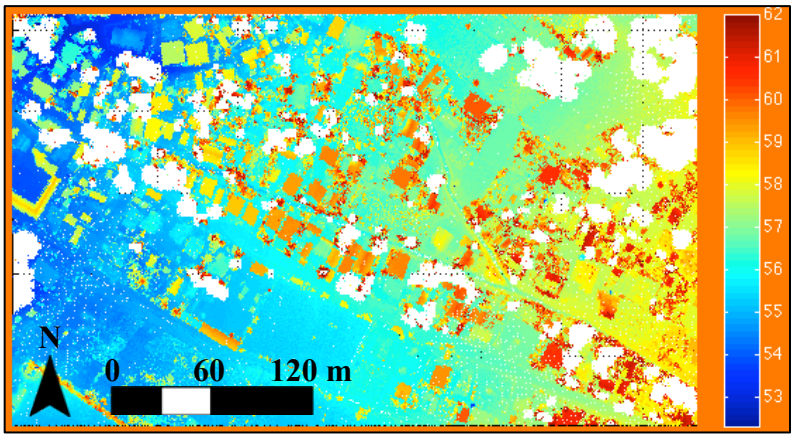


Figure 24. Raw point cloud after points greater than 62 m were removed.

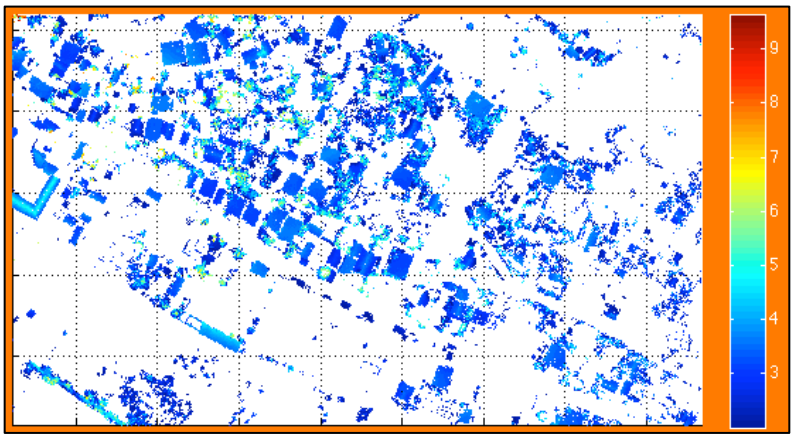


Figure 25. The LiDAR point cloud, referenced above ground, with points below 2 m removed.

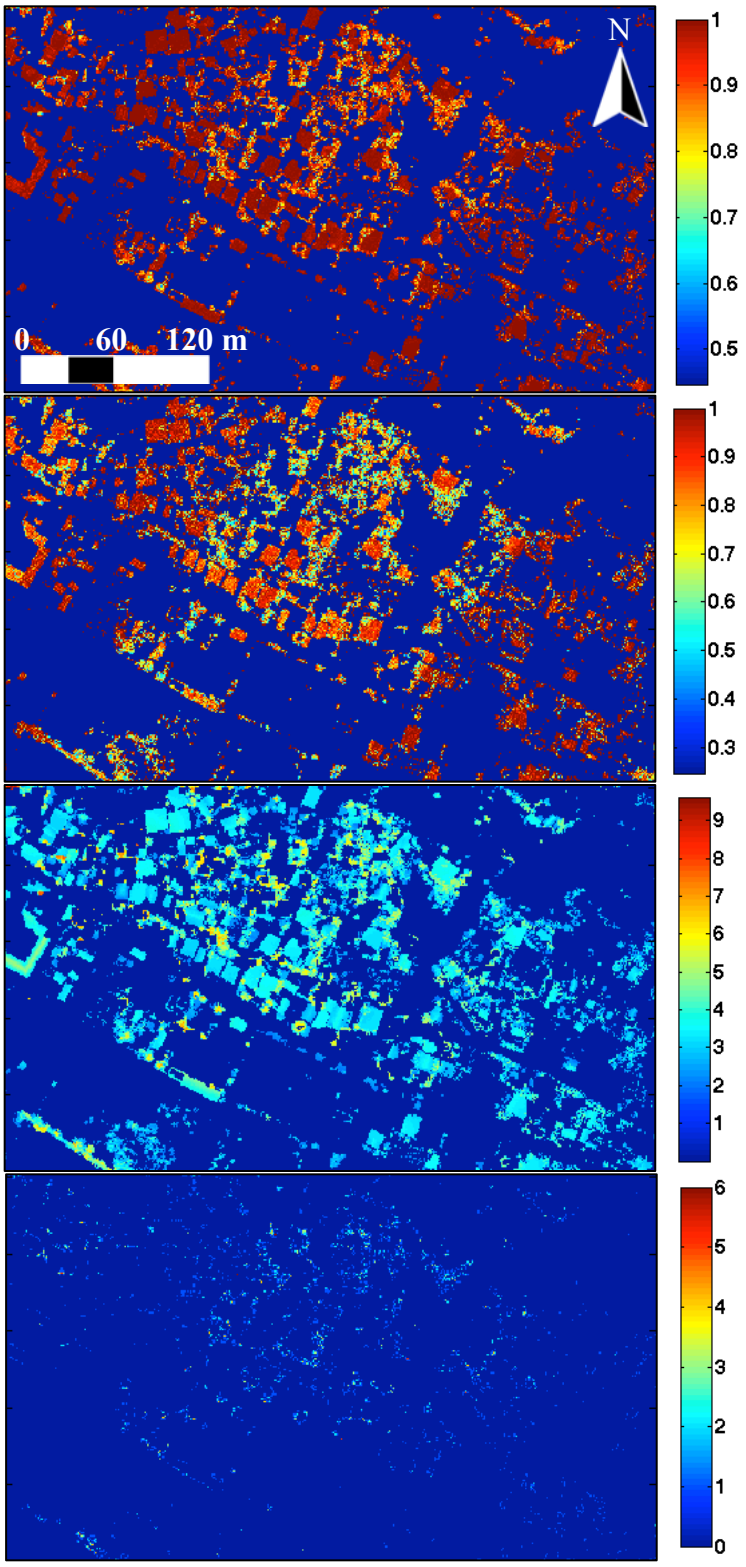


Figure 26. Images portraying different metrics that can be used for building segmentation. *Top to bottom:* normalized height, normalized intensity, maximum height, and multiple returns.

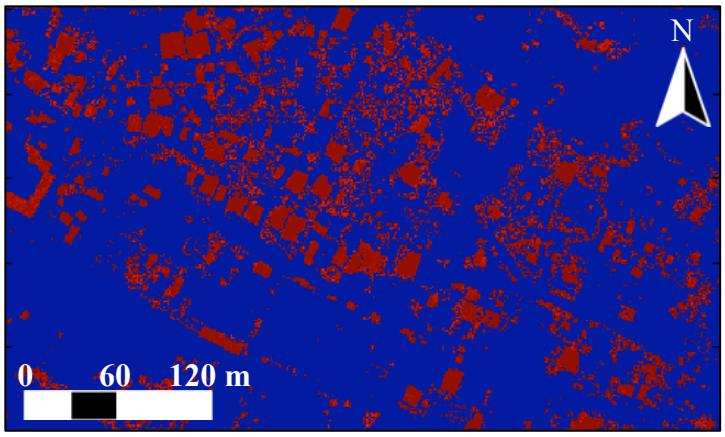


Figure 27. Normalized height image with pixel values less than 0.9 removed.

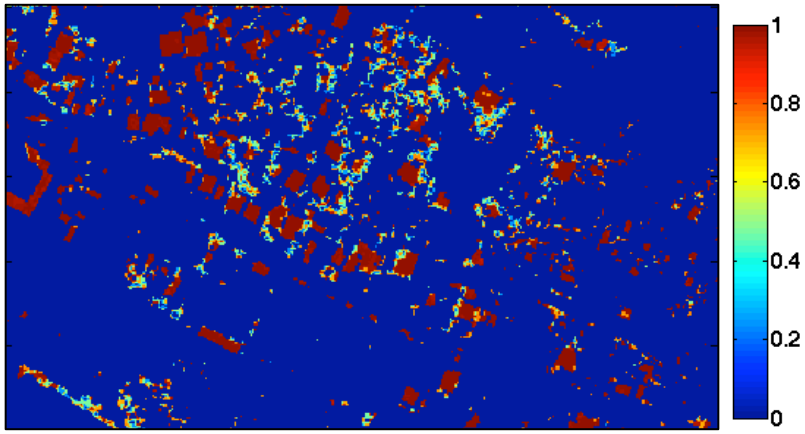


Figure 28. Result of using a Gaussian lowpass filter, with a 2 x 2 kernel, on the image in Figure 27.

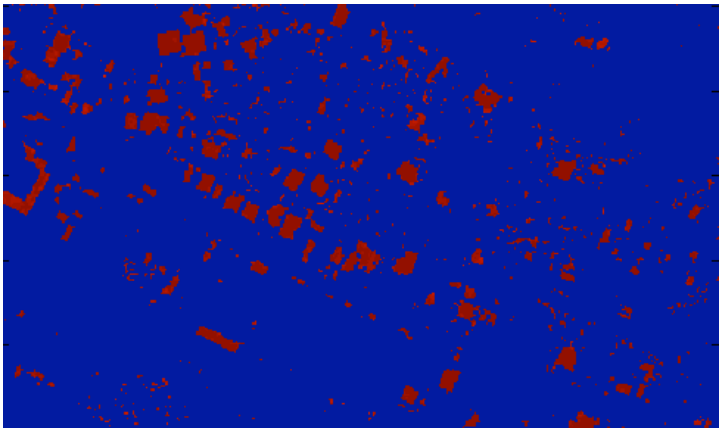


Figure 29. Result of applying a normalized height threshold of 0.8 to Figure 28.

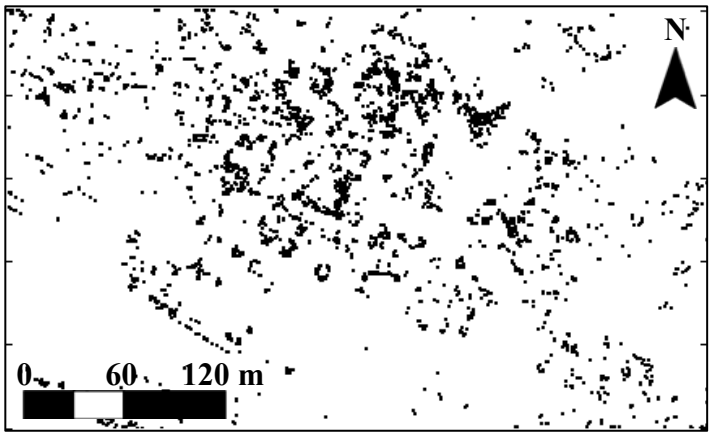


Figure 30. Binary multiple return mask created by applying a Gaussian lowpass filter to the multiple return image. Black pixels correspond to areas with more than one multiple return, which is indicative of vegetation.

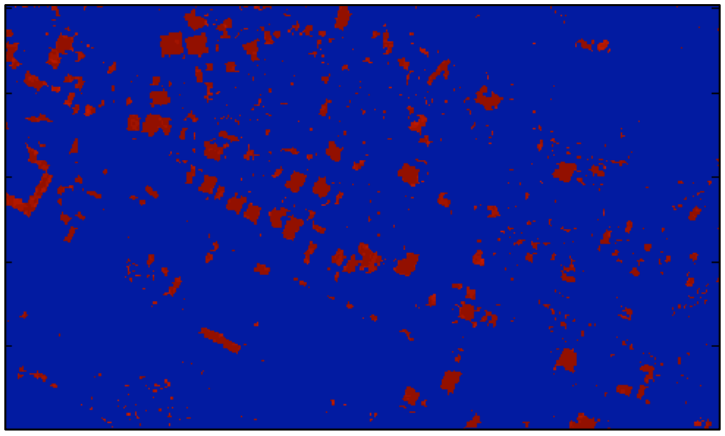


Figure 31. Result of masking out the areas with more than one multiple return from the initial building map in Figure 29.

The remaining building detection tasks were concerned with finalizing the segmentation and identifying unique building regions: The user could essentially perform morphological image processing operations on the binary building map to refine the building regions. Three common morphological operations were built into the tool: erosion, opening, and closing. The operations could be combined with either a square or an ellipsoid structuring element.

Erosion shrinks objects in a binary image, and can be used to split a single object into multiple objects. It works well at removing thin connections between larger objects,

which is useful when trying to separate two obviously unique, yet connected building regions (Gonzalez *et al.*, 2009). The processes work by translating a structuring element over the building map, during which areas in the map that cannot fully contain the structuring element are removed. In other words, the structuring element cannot overlap any of the image background.

Morphological opening is defined as an erosion followed by a dilation, while closing is the opposite, a dilation followed by an erosion. A dilation grows an object, by translating a structuring element across an image, and placing a "1" at each location of the origin of the structuring element if the structuring element overlaps at least one, 1-valued pixel in the image (Gonzalez *et al.*, 2009). Opening and closing are both useful on noisy images and tend to smooth object contours. Opening is used to remove thin connections and thin protrusions, while closing tends to join narrow breaks and fill holes smaller than the structuring element.

In the case of the Darbonne scene, there were many noisy pixels that did not represent building regions. These were removed by performing a morphological opening using a square structuring element that was two pixels wide. Essentially, any objects or "blobs" of pixels that were smaller than 2 x 2 were removed. The result of the opening is shown in Figure 32.

The last step in the building detection workflow was to extract individual building regions. This was accomplished by analyzing the connection types between pixels, or in image processing terms, computing the connected components. The approach used in this research looked for 4-connected objects, meaning as long as a pixel shared a side with another pixel the two were grouped in the same component. The pixels in each

different connected component, or object, were assigned a unique label. The final building map of the Darbonne scene is shown in Figure 33. There were 205 total building regions identified, each shown in a different color. As a comparison, there were 258 GEO-CAN assessments recorded for Darbonne. A complete accuracy assessment is provided in Section 5.2.

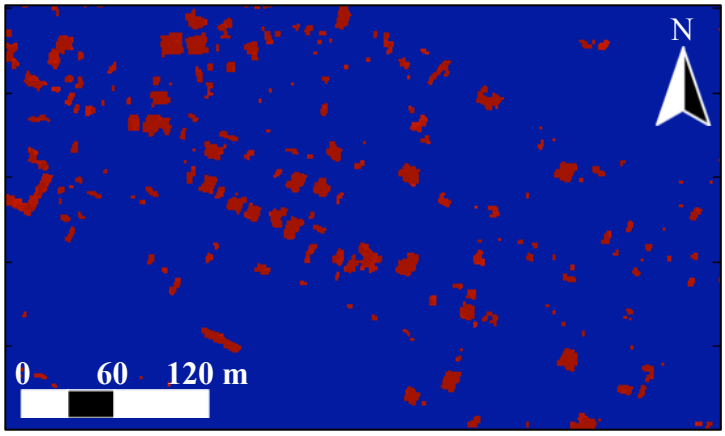


Figure 32. Building map after morphological opening using a two-pixel wide square structuring element.

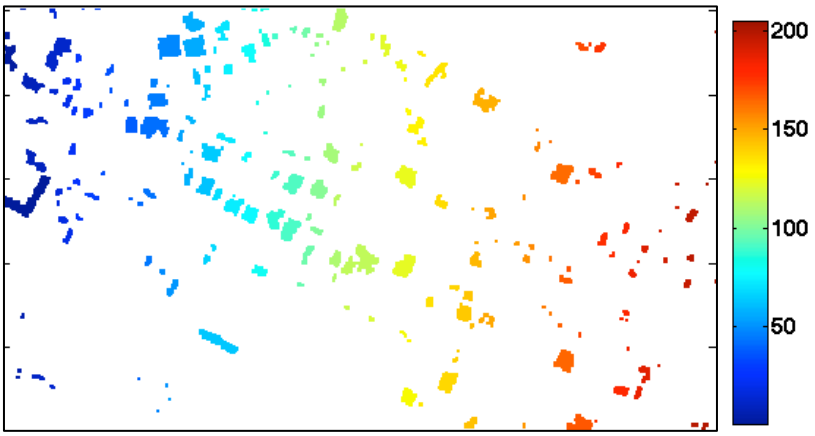


Figure 33. Final building map of Darbonne showing the 205 building regions detected. Each unique building region is shown in a different color.

4.5 Building Damage Detection

The building regions identified using the methodology discussed in Section 4.4 were the inputs to the damage detection algorithm. For each building region identified,

the height model LiDAR points located within the region were extracted. The resulting region-specific point clouds consisted primarily of building roof points, where roof planes were dominant features. Three building damage techniques were considered to detect damaged building points. These techniques were based on the assumption that roof points would not fall on the same plane if the structure below were damaged. Unless there was a perfect pancake collapse, where the roof maintained its shape entirely throughout the building area, roof points would generally appear displaced when a building was damaged.

The first technique tiled the building points and used principal components analysis (PCA) to find the normal vector for each tile. The vectors were then compared to identify outliers, or the tiles whose points did not fall on the dominant roof plane. This method started by tiling the building points according to a user-defined tile size. PCA was then performed on the set of points for each tile. PCA is an orthogonal linear transformation that transforms data to a new coordinate system in such a way that the projection of the data onto the first principal component axis exhibits the most variance, or information. The second principal component axis is orthogonal to the first, and has the second greatest variance, and so on. PCA is designed to decorrelate multi-dimensional data and maximize the variability in a reduced number of dimensions (Schott, 2007).

PCA is often used to fit a plane to 3D data. The coefficients for the first two principal components define vectors that form a basis for the plane, while the coefficient of the third principal component defines the normal vector of the plane. To compare the normal vectors for all the tiles, the angle between each normal vector and a reference,

zenith vector was computed. The resulting set of angles was then analyzed for potential outliers. This process will be described in depth, using the building point cloud shown in Figure 34 and the corresponding histogram of the normal angles in Figure 35.

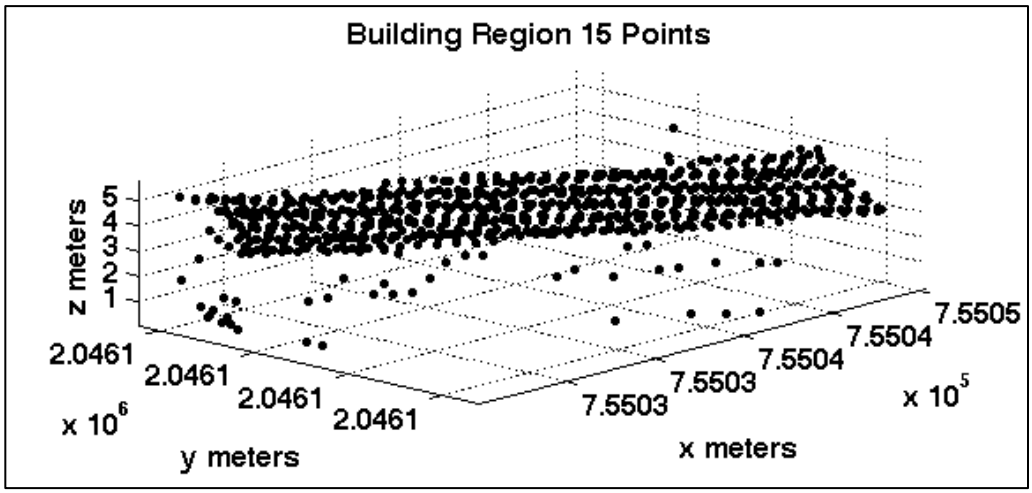


Figure 34. Point cloud of a building region, extracted from the Darbonne scene.

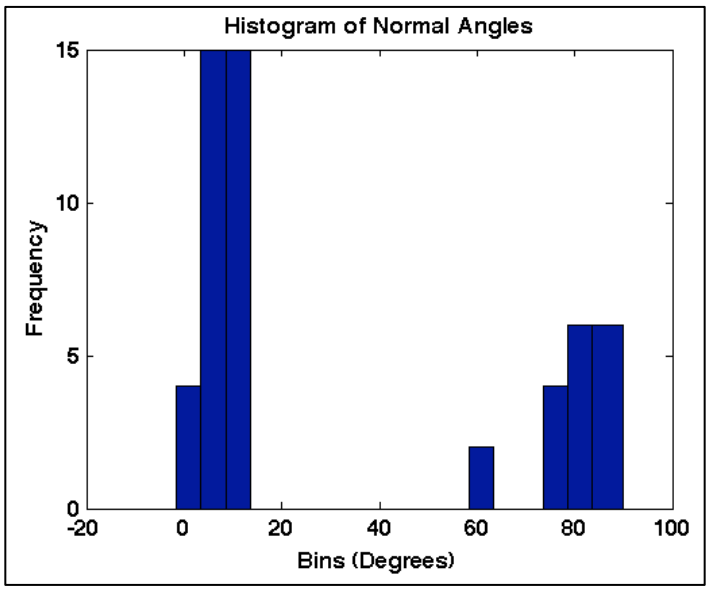


Figure 35. Histogram of the angles between the normal vector to every tile plane and a reference zenith angle. The bins range from 0-90° and are 5° wide.

The histogram of the angles provided insight into the number and orientation of dominant planes in the building point cloud. If a building has a completely flat, undamaged roof, the histogram of the normal angles should have all the data residing in a

single bin. Damaged roofs, however, are expected to have a number of random angles, and therefore more bins will be filled and dominant planes will be harder to detect. The histogram in Figure 35 indicates that the majority of the angles, 66% to be exact, fall within 0-13.5°. These angles represent the dominant plane in Figure 34, which means their corresponding tiles likely do not contain damaged points. The remaining third of the angles, however, appear to be caused by tiled points that do not share a plane, even though this is difficult to distinguish in Figures 34 and 35. It is evident that determining how many dominant planes are present in the data from the histogram of normal angles could be a challenging task. Looking purely at the number of distinct peaks in the histogram distribution did not always provide accurate results, since many of the Haitian buildings are small and therefore only have a limited number of tile angles to consider. In these cases, the distribution might not have any peaks, or at least none that are distinct.

An assumption was made that if 20% or more of the tiles had angles within ±2.5° of each other, then these tiles lie on the same plane. This allowed multiple planes to be considered, by thresholding the histogram frequencies, while limiting confusion due to local maxima. The building point cloud in Figure 34 was divided into 52 total tiles, therefore histogram bins with frequencies of 11 or more were assumed to be the "undamaged" angles. Since the remaining angles did not appear to represent a plane, they were considered "damaged". A simple damage metric was then computed by dividing the number of "damaged" angles by the total number of angles.

Figure 36 shows the building point cloud with the normal vectors extending from each tile. The "undamaged" vectors, shown in blue, represent similar angles and therefore points that lie on the same plane. Conversely, the red or "damaged" vectors

correspond to points that do not share a plane. The damage percentage for this building region was 42%. The detected damage in this case was attributed to a segmentation artifact, where low, outlier points were included in the building region and considered by the algorithm, despite the fact that they did not belong in the building point cloud. It was decided not to filter out these points and try to improve the results, since perceived segmentation errors also could be due to damaged, collapsed points. It is important to note that all the damage detection techniques assumed that the building point clouds consisted of solely building points – their accuracy therefore was directly related to the accuracy of the building segmentation.

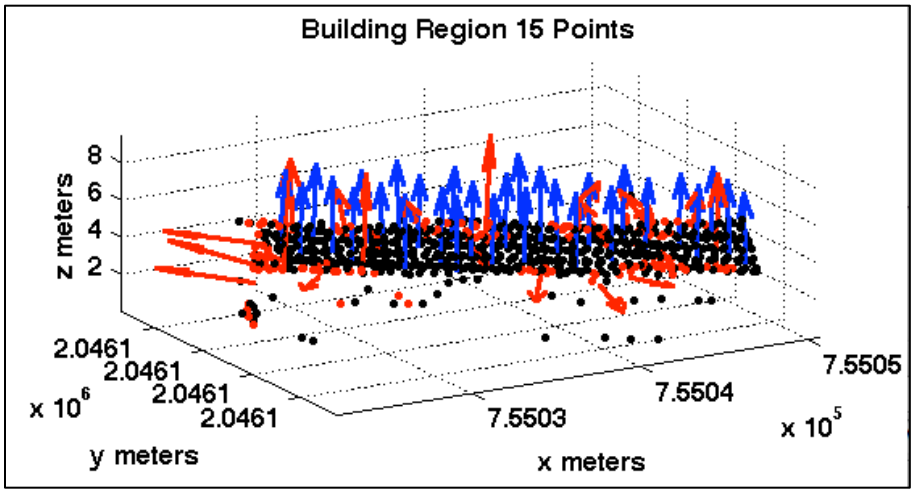


Figure 36. Building point cloud with the normal vectors shown for each tile. The blue vectors correspond to undamaged points, while the red vectors indicate damage. According to the normal angle metric, 42% of the building points are damaged.

The second technique to detect building damage was also based on tiling the building point cloud, but instead calculated the variance of the points in each tile. This technique was very similar to the variance-based method that was used to find vegetation points in the first building segmentation step. It assumed that in a local area, undamaged

points would have very little height variance, while points with a large amount of height variance were probably indicative of damage.

The approach started by dividing the building region point cloud into tiles of a user-specified size. The variance of the heights of the points in each tile was then computed and the resulting set of variances for all the tiles was analyzed for large values, or outliers. At first, variance values were deemed "large" based on the mean and standard deviation of the set, but this assumed a normal distribution. An examination of the variance histograms showed that the distribution of the variances looked more like an impulse function, with the overwhelming majority of values residing extremely close to zero. Tiles with undamaged points seemed to always have variances less than 0.01 m^2, and therefore were usually combined in the same, large bin. As point heights started to deviate from a plane, the variance grew exponentially, and therefore it was easy for large variance values to influence the mean and standard deviation of the distribution. To overcome this issue, large variance values were defined as those above 0.03 m^2.

Figure 37 shows the histogram of tile variances calculated from the building point cloud displayed in Figure 34. The threshold of 0.03 is marked in red on the *x*-axis of the histogram. Again, the point cloud was divided into 2 m-wide tiles, resulting in approximately 13 returns per tile. Of the 52 tiles in the scene, 19 had variances above the threshold. The points in these outlier tiles were all flagged as damaged, and divided by the total number of points in the building region point cloud to compute a damage percentage. Figure 38 shows the point cloud with the damaged building points highlighted in red. In this case, the damage percentage calculated was 30%.

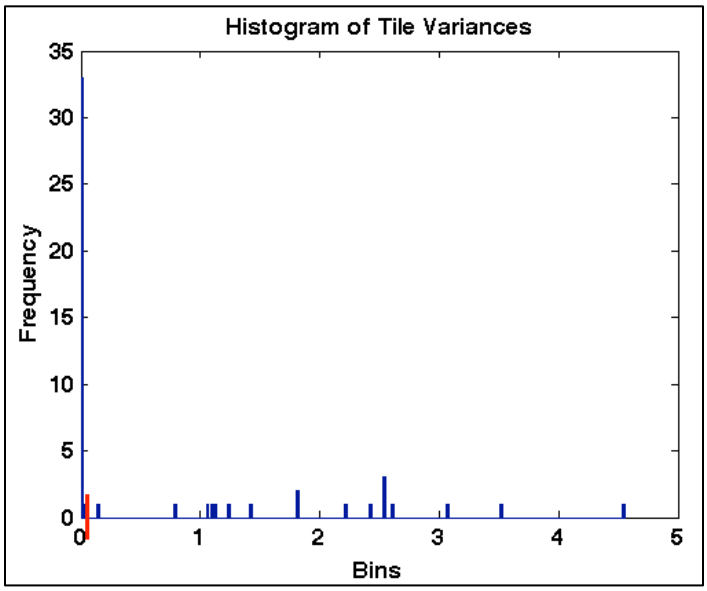


Figure 37. Histogram of the height variances of tiled points within a building region point cloud. Of the 52 tiles, 33 have variances below the 0.03 threshold, shown in red. The 19 tiles with larger variances are likely to contain damaged points.

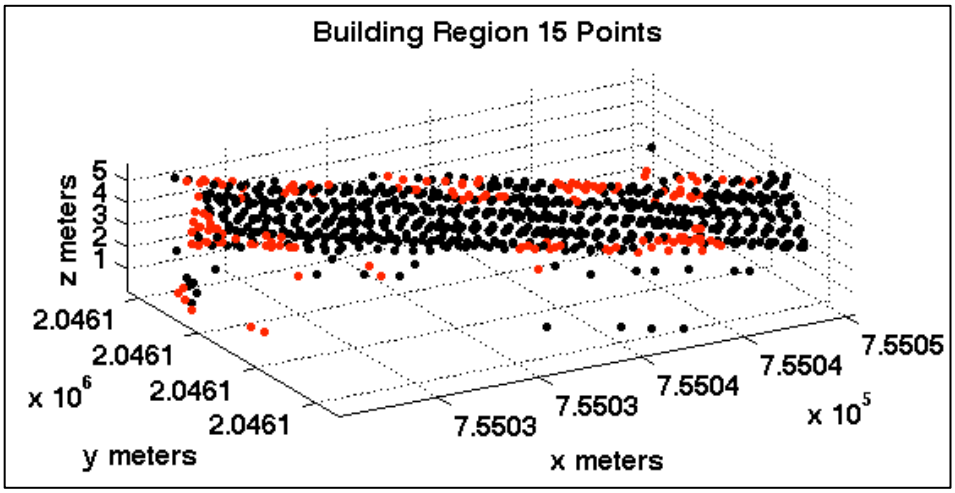


Figure 38. Building region point cloud with damaged points, as defined by the variance metric, shown in red. According to the metric, 30% of the building points are damaged.

The third technique developed to assess building damage was essentially a line-by-line scan for slope differences. The approach started by rounding the *y* coordinate for each point to the nearest tenth, in an effort to co-locate or bin the data. For each line of data, the slopes were then calculated between subsequent points in the *x* direction. The slopes, computed by dividing the change in height by the change in *x*, would be constant

on either side of a point if that point was undamaged. However, if a point was damaged, the slopes would differ on either side. This concept is illustrated in Figure 39. A user-defined threshold was implemented to determine whether the difference in slopes was great enough to classify a point as damaged.

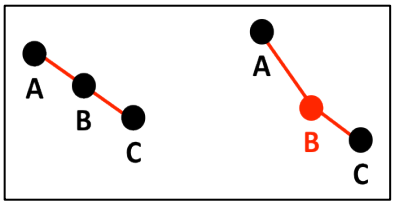


Figure 39. The slopes between consecutive points can be used to detect damaged points. In the set of points on the left, the slopes between points A and B and points B and C are constant, so point B is undamaged. On the right, the slope between points A and B is much steeper than the slope between points B and C, so in this case it is assumed that point B is damaged.

The same steps were repeated in the *y* direction. Figure 40 shows a nadir view of a building point cloud, with the "scan lines" in the *x* and *y* directions that were followed to compute the slopes between points. The union of the damaged points flagged from each perspective resulted in the total set of damaged points in the building region. The damage percentage metric was then calculated by dividing the number of damaged points by the total number of building points. The result of applying this technique to the point cloud, using a slope difference threshold of 0.1, is shown in Figure 41. Of the 762 points in the scene, 22% were assessed as damaged.

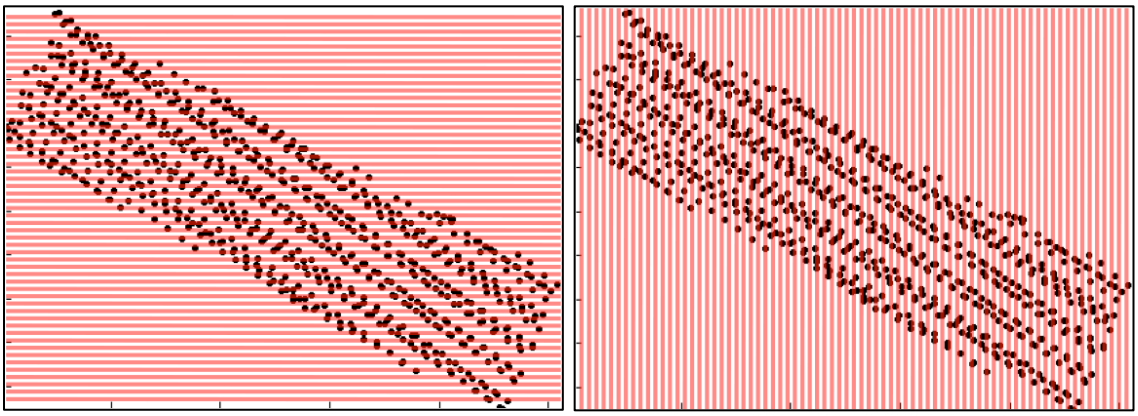


Figure 40. Nadir view of a building region point cloud. The scan lines followed to compute the slope differences are overlaid in the *x* direction (*left*) and *y* direction (*right*).

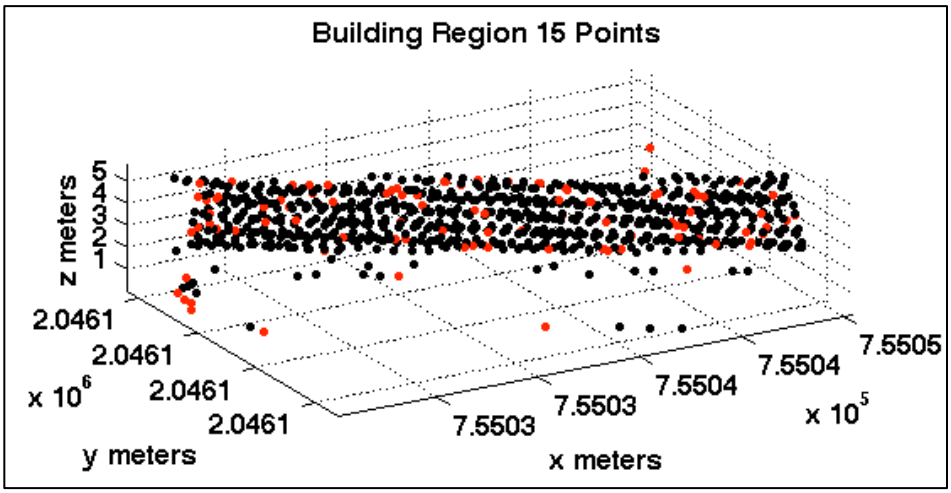


Figure 41. Building region point cloud with damaged points, as defined by the line-based slope difference metric, shown in red. According to the metric, 22% of the building points are damaged.

In order to create a damage map that would be useful to analysts and emergency managers on the ground, the three distinct damage detection metrics needed to be effectively combined into one overall damage metric. To accomplish this, the three techniques were initially evaluated and assessed for accuracy, using truth data from the GEO-CAN effort. The idea was that weights, or the amount that each metric contributes to the final damage assessment, would be assigned based on the performance of the different techniques.

The three methods under consideration were applied to a set of 30 building regions extracted from the Darbonne scene. The GEO-CAN assessment provided a damage rating for each of the buildings, based on interpretation by structural engineers and image interpreters. Grade 1 buildings had no visible damage, Grade 3 buildings suffered substantial to heavy damage, Grade 4 buildings were very heavily damaged, and Grade 5 buildings were completely destroyed. Feature visualization was then performed, where each of the buildings was plotted as a point in the three-dimensional metric space. The points were colored according to the truth damage level assessed for the particular building, extracted from the GEO-CAN data. The resulting 3D point cloud was manually rotated, with the hopes of gaining insight from the structure of the point cloud. Unfortunately, the points in the cloud were scattered and did not exhibit distinct clusters.

Combinations of two metrics therefore were plotted against each other to see if any additional information could be gleaned, since it appeared that the three metrics did not relate to each other well at all. These plots are shown in Figure 42. Out of the three possible two-metric combinations, the only plot that exhibited some form of clustering or correlation was the one relating the tile variance metric to the normal angle metric. The slope difference metric did not appear to be correlated to either of the other two metrics. Though correlation does not necessarily indicate accuracy, it is indicative of the relationship between the two metrics, which can be exploited. For instance, if it is not possible to use one metric for any reason, e.g., there are not enough building points to compare tile normal angles, the variance metric can be confidently substituted. Finally, just because there is not a statistical relationship between the slope difference metric and the other two metrics, it does not mean the metric should be discarded. It is possible that

the technique provides complementary information that is of use to the damage assessment effort. The true test to assess the accuracy and utility of the three metrics is to compare the truth damage grades with their corresponding damage percentages. After visual assessment of the plots, it was decided that the normal angle metric was the best predictor of building damage. Although no one metric seemed distinctly better than the others, it was evident that as the normal angle percent damage increased, more yellow and red points persisted. These colored points, which represented the highest grades of building damage, were the most crucial to classify correctly in terms of field response.

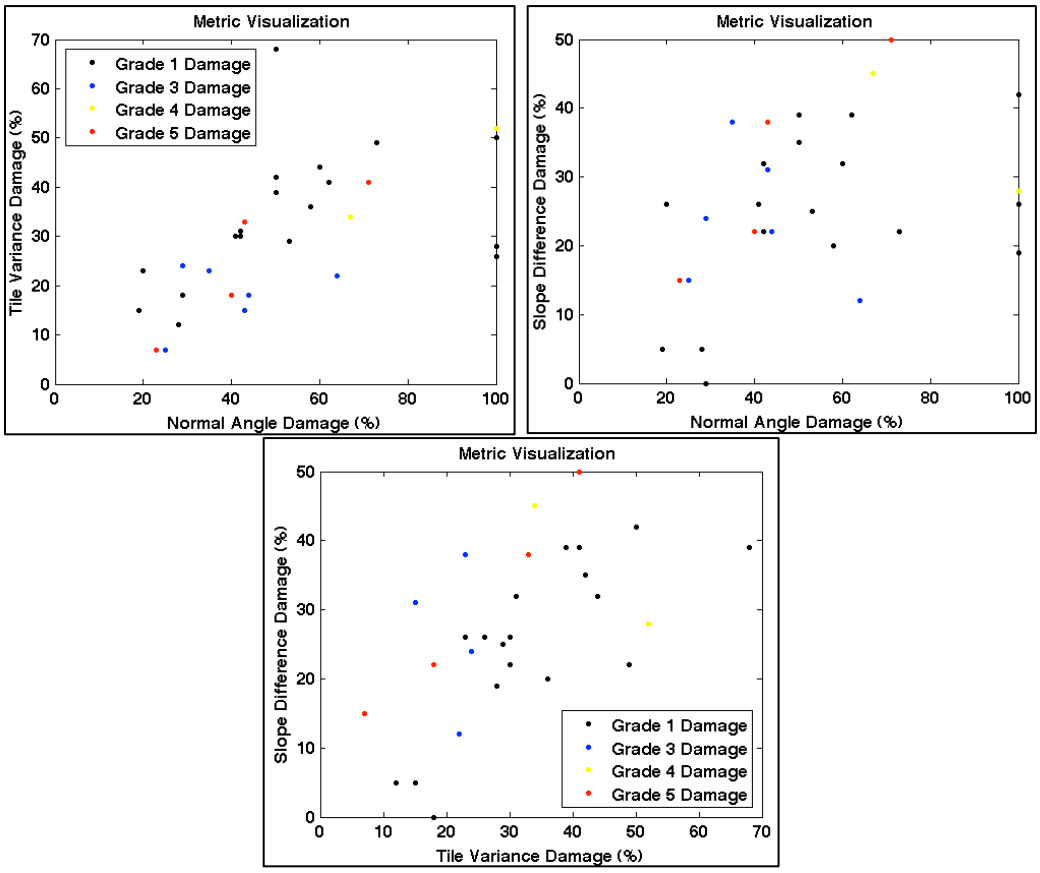


Figure 42. Plots comparing combinations of the three damage detection metrics. The points represent building truth from the Darbonne scene.

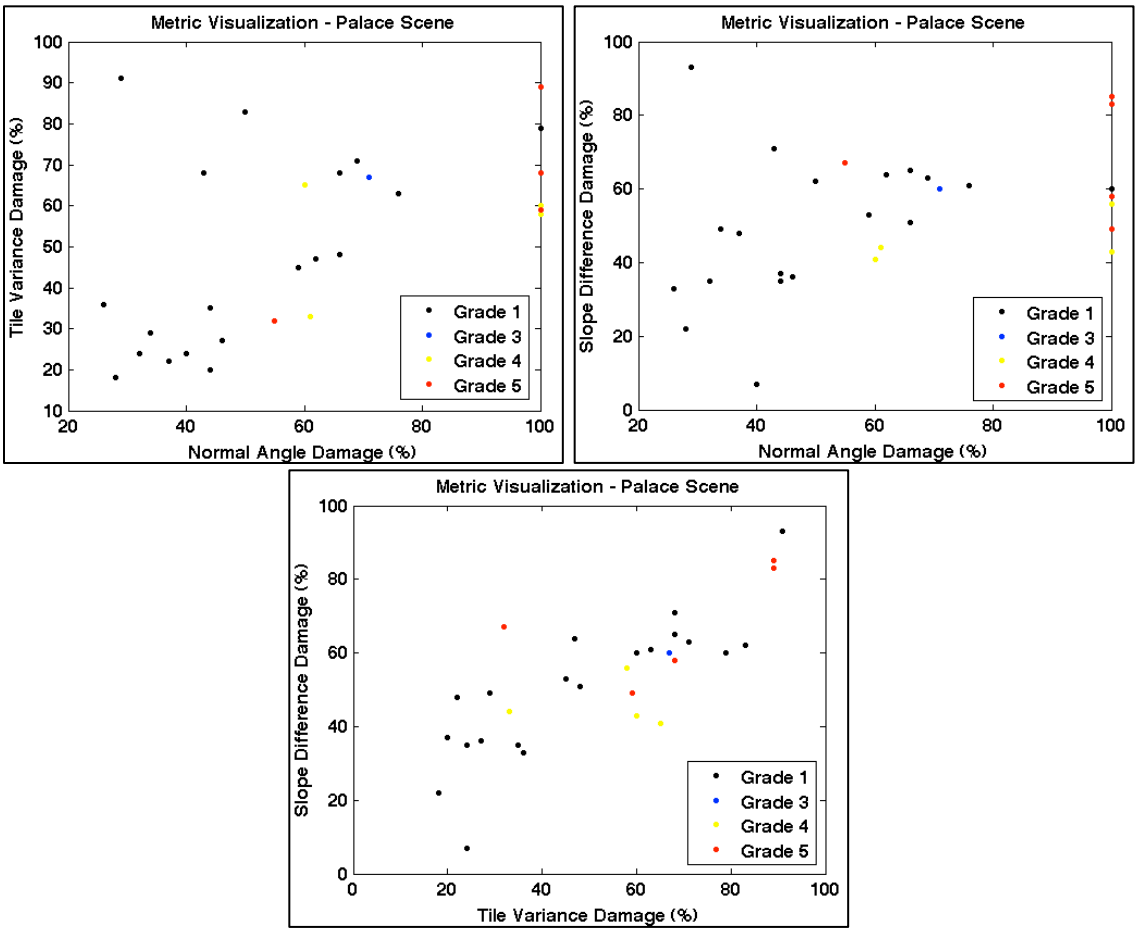


Figure 43. Plots comparing combinations of the three damage detection metrics. The points represent building truth from the scene surrounding Haiti's National Palace.

The same steps were carried out using building truth from the scene surrounding Haiti's National Palace. The resulting plots are shown in Figure 43. The correlation between the tile variance and normal angle metrics was not as evident in this case, although slope difference and tile variance did seem to be correlated or clustered according to truth damage data. Judging from the plots, the normal angle metric again appeared to be the best predictor of building damage. Although in this case the tile variance and slope difference metrics did a better job of classifying the Grades 3-5 buildings than they did in the Darbonne scene, the Grade 1 buildings were spread across the range of damage percentages.

It therefore was decided to use the normal angle technique, almost entirely by itself, as the overall damage detection metric. This was a confident choice, not only on the analysis presented here, but also based on the hours of development and testing of the three metrics. Countless building point clouds were examined using the techniques and the normal angle metric always seemed to best represent the level of damage. The variance metric worked relatively well, but tended to be sensitive to small height variations that should have been ignored. The least amount of confidence was placed in the slope difference metric, as it seemed to arbitrarily flag undamaged points as damaged. This was attributed to the way the points were binned before calculating the slopes. If the slopes could be computed between subsequent points along the natural laser scan line, results should be greatly improved.

Initial results showed that although the normal angle damage detection technique generally performed well, there were some situations in which it failed. As alluded to earlier, the buildings in Haiti range from small, shanty-type structures to larger structures like the presidential palace. In some cases, the building point clouds were so small that they could only be divided into a couple of tiles. Comparing two normal vectors did not provide accurate damage information, so in these situations another approach should be considered. In addition, there were cases in which the truth data indicated that a building was destroyed, but the building point cloud appeared to be a uniform plane of points, with heights near ground level. Comparing the normal vectors would indicate that the building was undamaged. If the point heights were taken into account, however, the damage could be properly assessed.

In order to address these situations, two rules were added to the overall damage detection metric. First, if there were less than 50 total points in a building point cloud, or less than four useable normal vectors, the variance metric was used to calculate the damage percentage in place of the normal angle approach. Second, if 90% or more of the total number of building points had heights below 1 m, the building was automatically classified as being destroyed, or 100% damaged. The refined damage detection approach produced the building damage maps of the National Palace and Darbonne scenes shown in Figure 44. The building segments have been colored according to their percent damage.

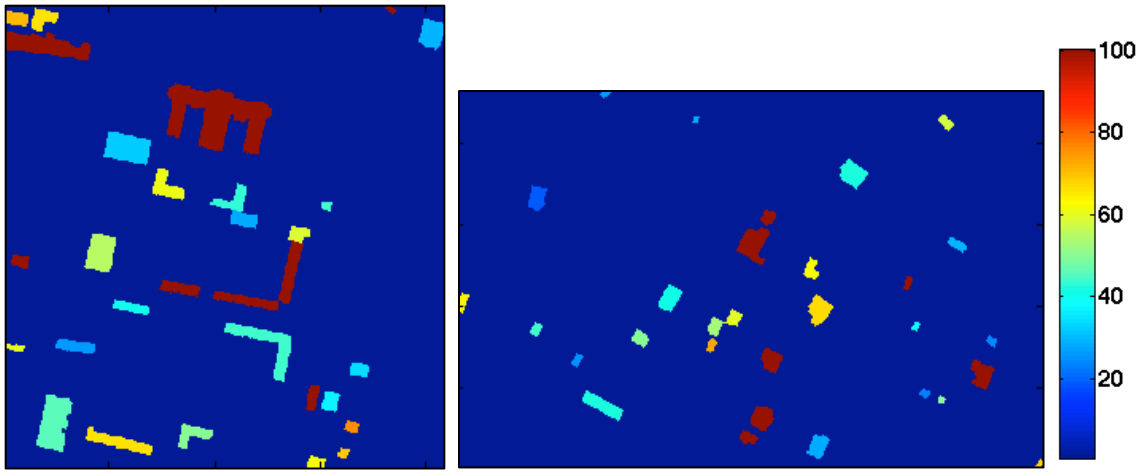


Figure 44. Building damage maps of the scene surrounding Haiti's National Palace (*left*) and Darbonne (*right*). GEO-CAN assessments in this scene range from Grade 1 (undamaged) to Grade 5 (destroyed).

The resulting damage percentages could then be compared to the GEO-CAN truth data to assess the accuracy of the semi-automated, LiDAR-based approach. This was done initially for the two scenes shown in Figure 44. It was determined that a threshold of 51% best separated the undamaged buildings from the buildings with damage grades 3-5. A binary decision was selected to distinguish between two classes; undamaged and

damaged, instead of binning the percentages to provide higher resolution in building damage classes. The binary decision was necessary due to the fact that as the GEO-CAN assessment grade increased from three to five, the corresponding percent damage did not always exhibit a similarly consistent response.

When the threshold was implemented on the scene surrounding Haiti's National Palace, 73% of the 30 truth buildings were classified correctly, while the remaining 27% of the buildings were said to be damaged, but were actually undamaged. Damage was underestimated in none of the buildings. The damage detection algorithm performed worse in the Darbonne scene, as it only correctly classified 57% of the buildings. Of the remaining buildings, 30% were overestimated to be damaged, while in the remaining 13%, damage was underestimated. Examples of quick damage assessment maps that were generated by thresholding the damage percentages are shown in Figure 45. A more thorough analysis and discussion can be found in the Results section.

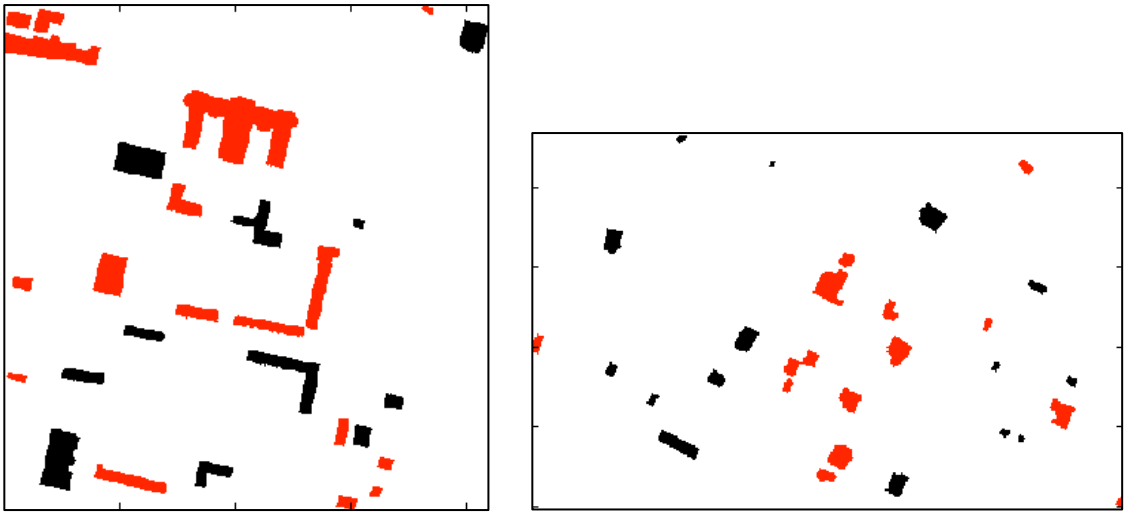


Figure 45. Example damage assessment maps that could be given to emergency managers or responders on the ground following a disaster. The buildings assessed to be highly damaged are shown in red, while building in black are assumed to be undamaged.

4.6 Validation

It was obvious from the early stages of this research that the varying building and terrain environment in Haiti would present challenges in terms of size, shape, and building materials and slope, respectively. In the region surrounding Port-au-Prince that was devastated by the earthquake, buildings range from large concrete structures to small wood and metal shanties. In addition to the vast differences in building footprints and materials, the terrain also changes between rugged mountains, coastal plains, and river valleys.

In order to test the robustness of the algorithms, six validation sites were chosen that reflected the varying building and terrain environments in Haiti. Sites were selected based on building density per area and average slope of the terrain. The four types of validation sites were: sparse/flat, dense/flat, sparse/steep, and dense/steep. Sites were chosen that aligned with these categories and included the National Palace and Darbonne scenes that were used in algorithm development.

"Dense" refers to sites that had more than 30 assessments per hectare. Assessments corresponded to the GEO-CAN truth data, and generally referred to a single building, though that was not always the case. GEO-CAN sometimes recorded multiple assessments for larger buildings. The number of GEO-CAN assessments in an area provided a decent estimate for the number of buildings, without requiring the buildings to be manually counted. "Sparse" sites were those with 30 or less assessments per hectare. Thirty was chosen as a threshold by comparing a few obviously different building density sites. Residential areas with buildings almost on top of each other had assessment densities of greater than 50 per hectare, while significantly less dense areas by

observation had assessment densities in the mid-twenties and below. Thirty assessments per hectare were determined to be a natural cutoff.

The average slope of the terrain was calculated from the Digital Terrain Model (DTM) of each site. Once extracted from their raw point clouds using the technique described in Section 4.3, the raster DTMs were imported into ESRI ArcMap 9.3, a geospatial processing software program. As part of its *Spatial Analyst* toolbox, the program has a *Slope* tool that calculates the maximum rate of change in value for each cell in a surface in relation to its neighbors. A slope raster was created for each validation site. The mean slope value was then computed for each raster as a metric to compare the terrains between sites. An example slope map for the Darbonne terrain, which has an average value of 1.66°, is shown in Figure 46. Sites were considered "steep" if their mean terrain slope exceeded five degrees. This cutoff value was again chosen and refined based on observations. Rugged mountain areas typically have average terrain slopes of 10° and above, while flat coastal plains generally have average slopes below three degrees.

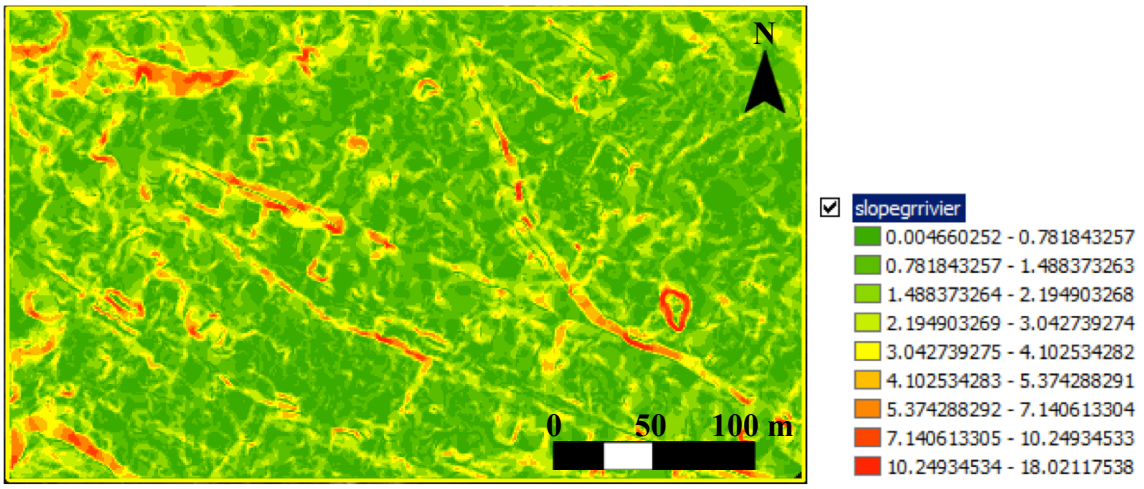


Figure 46. Slope map in degrees of the Darbonne terrain calculated by ArcMap. The average slope of the scene is 1.66°, so the terrain would be considered "flat".

Six validation sites were selected that not only fit the four categories, but that also had truth data available. The GEOCAN damage assessment was the primary truth data source; however, the Earthquake Engineering Field Investigation Team (EEFIT) provided some additional truth data collected during a ground assessment. The EEFIT data, though not nearly as extensive in coverage as the GEO-CAN assessment, was far more accurate and descriptive. Figure 47 shows the validation site "Turgeau", located in Southeast Port-au-Prince. The blue and yellow dots on the image correspond to truth assessments performed by GEO-CAN and EEFIT, respectively. The six validation sites are listed in Table 2, and colored according to category. For each validation site, the corresponding point density, assessment density per hectare, and mean terrain slope are shown. Maps detailing the locations of the validation sites are included in Appendix A.

Thirty buildings were randomly selected in every validation site to serve as truth for evaluating building segmentation and damage detection performance. Truth building masks were manually created in ArcMap by tracing the building outlines from the high-resolution WASP imagery. The building outlines were overlaid on the output building

map to detect overlap and assess accuracy of the segmentation routine. They were also used to "cut out" or mask the height model point cloud to provide truth building regions to the damage detection algorithm.

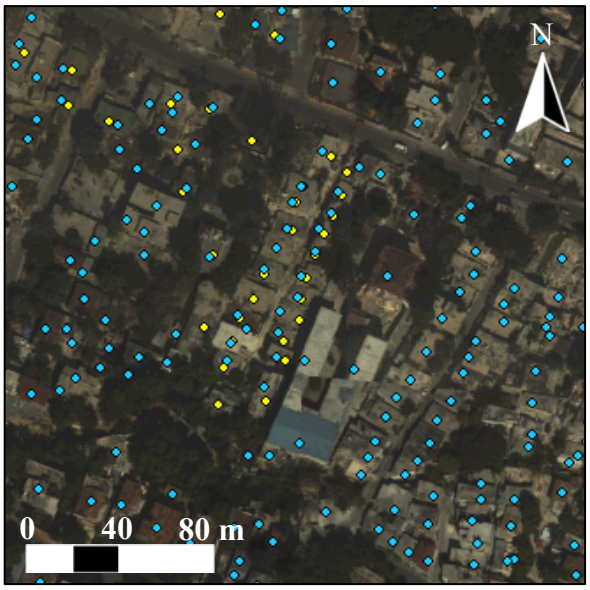


Figure 47. Image of the Turgeau validation site located in Southeast Port-au-Prince. The blue dots correspond to GEO-CAN damage assessments, while the yellow dots represent EEFIT evaluations.

Table 2. Six validation sites categorized by building density per area and mean terrain slope. The point density for each site is also listed.

Validation Site	Hit Density (points/m^2)	Assessment Density (per hectare)	Mean Terrain Slope (degrees)
Palace	4.38	5.58	2.77
Darbonne	3.47	24.15	1.66
Grand Goave	4.48	21.09	2.31
Léogâne	1.83	64.71	1.78
Turgeau	4.46	26.03	5.64
Riviere Froide	4.54	66.22	14.04

Dense: > 30 assessments/hectare
Steep: > 5° average slope

Sparse/Flat | Sparse/Steep
Dense/Flat | Dense/Steep

4.7 Chapter Summary

This chapter described the essential tasks to develop an end-to-end operational tool that ingests a LiDAR point cloud of a post-disaster scene and outputs a building damage map showing damaged and collapsed structures. First, the effect of the terrain was removed from the raw point cloud to derive a normalized surface model. Second, the building points were separated from the rest of the point cloud using a combination of point classification techniques. Lastly, damage was detected by measuring the deviation between building roof points and dominant planes. The methodology described in this chapter was implemented in a Matlab graphical user interface (GUI) that resembles the final tool. With the process defined and the tool created, the next step was to use the validation approach to evaluate performance.

5 Results and Discussion

5.1 Chapter Overview

Results for the entire approach are provided for a set of validation sites, carefully selected from the Haiti LiDAR data. For each site, the performance of the building segmentation and damage detection algorithms are assessed. The strengths, weaknesses, and limitations of the algorithms are discussed. The chapter will conclude with some recommendations for future work.

5.2 Building Segmentation Performance

To evaluate the building detection workflow, the tool was interactively run on each of the validation sites, and the output building maps were compared to the building truth sets. Throughout the process, the parameters and operations were optimized for the particular scene being analyzed. A summary of the results for each of the validation sites is presented in Table 3. It is important to note that since only 30 buildings from the entire scene were used for validation, this method only tested for false negatives, not false positives. The output maps and the corresponding WASP imagery for all the sites are included in Appendix A.

Table 3. Summary of results from interactively running the building detection routine on each of the six validation sites. The number of building segments identified in the output building map is listed alongside the total number of GEO-CAN assessments for the site. The 30 truth building outlines were overlaid on the building map to determine the total number of buildings detected, and how many went undetected.

Validation Site	Total Building Segments	Total Building Assessments	Total Buildings Detected*	Total Buildings Undetected*
Palace	168	115	29	1
Darbonne	205	258	20	10
Grand Goave	182	279	21	9
Léogâne	325	295	24	6
Turgeau	186	164	30	0
Riviere Froide	288	296	30	0

* Out of the 30 truth buildings

Sparse/Flat Sparse/Steep
Dense/Flat Dense/Steep

While manually assessing the performance of the building segmentation algorithm for each of the six validation sites, several common themes emerged. First, there was an over-segmentation of the buildings in half of the sites. This was caused primarily by not being able to remove all the vegetation points, which eventually were confused with building points. Second, poor segmentation results were common in very dense building areas. Areas where buildings were dense on a local basis, i.e., on top of each other, were rarely properly segmented, even within sparse validation sites. Third, the algorithm had difficulty detecting small buildings. Buildings greater than 5 m x 5 m had a much better chance of being detected than those below that threshold. Finally, some of the truth buildings may not have been buildings at all. There were several regions that looked like collapsed buildings in the imagery and were assessed as damaged by GEO-CAN, but actually may just have been bare earth or livestock corrals. Each of these themes will be discussed in further detail, with supporting examples from the validation effort.

Building region over-segmentation was most evident in the result from the Palace scene, shown in Figure 48. In this case, the building segmentation algorithm identified 168 unique building regions, which was more than the 115 GEO-CAN assessments for the site. Although the workflow performed well to detect all but one of the 30 buildings, it did produce far more building segments than actually existed in the scene. Over-segmentation was primarily a result of vegetation points not being completely removed, and thus believed to be part of building regions. In addition, some over-segmentation was noticed in larger buildings, where the output map showed multiple regions within one building outline.

False alarm building regions caused by incomplete vegetation removal was evident in every site. The regions showed up as very small groups of pixels, e.g., 2 x 2 pixels, since that was the size of the erosion structuring element and appeared to be noise in the building map. These false alarm regions can be seen in the Palace results in Figure 48 and up close in a zoom-in of the Darbonne building map in Figure 49.

Conversely, over-segmentation also occurred as a result of operating on large buildings. The National Palace, for instance, was broken into five separate regions. Another longer building, shown in Figure 50, was separated into two segments. Both buildings were highly damaged, however, which clearly influenced the segmentation results. Some could argue that over-segmentation in these cases could actually be helpful given the primary research task, which is to eventually separate damaged structures from non-damaged ones. In the case of a relatively large building like the Palace, dividing it into sections could lead to a higher resolution and more accurate damage assessment.

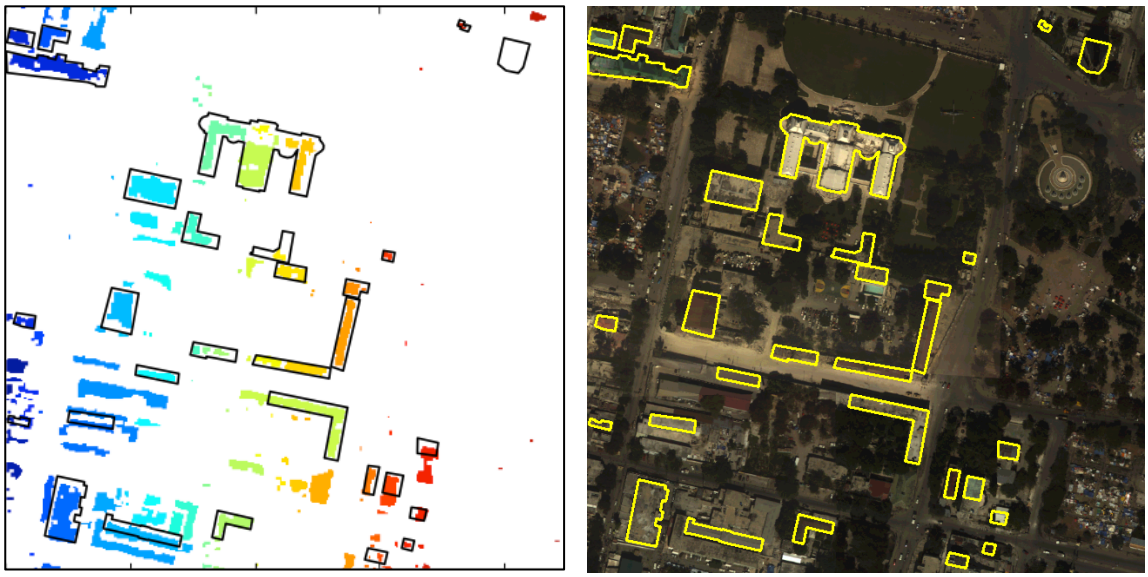


Figure 48. Truth building outlines for the National Palace validation site are overlaid on the output building map (*left*) and WASP imagery (*right*). Haiti's Bicentennial Monument, located in the top-right corner of the scene, was the only building not detected. The normalized height and normalized intensity metrics that are used to classify building points and look for uniform point areas, were hampered by the monument's tall and very steep sides.

Figure 49. Zoom-in area of the Darbonne validation site illustrating the effects of over-segmentation. Arrows point to building segments in the output building map (*left*) that do not correspond to actual buildings in the scene. The false alarms were caused by the thick vegetation that was not completely removed during the building segmentation process.

Figure 50. Zoom-in area of the Palace validation site where over-segmentation of a long building has resulted in two separate building regions. Close examination of the WASP image (*right*) shows that the building was heavily damaged from the earthquake, which influenced the segmentation results.

On the other hand, over-segmentation could lead to problems, since "false" buildings effectively are being created. In the context of the research problem, emergency managers are going to have to sift through all of the buildings, which can take time, a precious asset in disaster response. If the two images in Figure 48 are compared, many of the "noisy" building segments become obvious without too much effort. If additional context information, e.g., an image, is available to responders, the over-segmentation becomes less of a problem.

Another common issue that surfaced while analyzing the results was that the accuracy of the output building map suffered in areas of high building density. Too many buildings in a small geographic area, especially when they were hardly separated, caused problems with the building segmentation approach. The algorithm picked out the individual building roofs based on uniform height or intensity metrics, but the areas were too small and close together and were negatively impacted by the morphological operations. For example, roof structure was lost for the small areas when erosion and opening were performed to "clean up" the segmentation. This is illustrated in Figure 51,

where a section of the Palace validation site contains closely spaced buildings – it is clear that the building map does not accurately reflect the actual building footprints on the left side of the street. However, the larger more spread out buildings were much better detected on the right side of the street.

Figure 51. Section of the Palace validation site showing poor segmentation results for an area with a high concentration of buildings. All of the buildings were detected but the segments do not accurately represent the actual buildings.

This was evident at all of the sites, given that packed, dense buildings are common in Port-au-Prince. As expected, the two validation sites in the dense category exhibited the worst results. The building segmentation result for Léogâne, which had an overall assessment density of 65 buildings per hectare, is shown in Figure 52. Comparison of the building map to the visual imagery indicated that the segmentation was poor in areas that had a high concentration of small buildings. The buildings lining the road, for instance, were not properly represented by the segments in the building map. Although most of the buildings were detected, since the segments did overlap the building outlines, the number, size, and orientation of the segments did not accurately characterize the layout of the buildings. The poor segmentation in this site could be

partly explained by the low average LiDAR point density of 1.83 returns/m^2, which was less than half the value of other scenes, but is also due to the high building density.

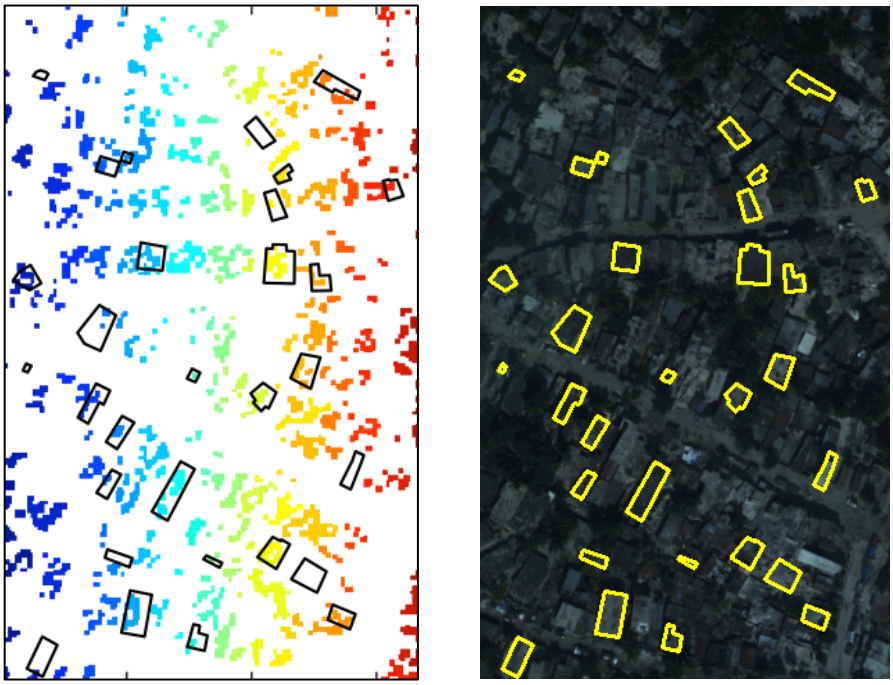


Figure 52. Truth building outlines for the Léogâne validation site are overlaid on the output building map (*left*) and WASP imagery (*right*). Though 24 of the buildings were detected, the segmentation results do not characterize the size, shape, and orientation of the actual buildings in the scene.

The results for Riviere Froide were similar, despite having two-and-a-half times the point density. In this case, every one of the 30 buildings was detected, but again the individual segments did not necessarily represent a single building. Figure 53 shows a roughly 50 m x 70 m area within the site, consisting of a high concentration of buildings. Very few of the buildings in this area, especially those lining the road, were individually identified in the building map.

Figure 53. Zoom-in area of the Riviere Froide validation site. Over 20 buildings exist in this 50 m x 70 m area, but were too small and dense to be properly segmented.

Poor segmentation had a knock-on effect in the workflow, since the building regions were used as inputs to the damage detection algorithm in an operational scenario: The detected building segments can theoretically be used to subset the height model point cloud, thereby providing a set of points for each individual building. The results would be skewed if a point cloud contained points from multiple buildings, since the damage detection routine is based on analysis of a single building at a time.

The third theme that was apparent while analyzing the building segmentation results was that the workflow had trouble detecting small buildings. In the Grand Goave validation site, shown in Figure 54, this shortcoming accounted for 8 out of the 9 undetected buildings. Buildings of less than 5 m x 5 m in size were rarely identified, due to the gridding of the point data to create the normalized height and intensity raster images. These small building areas became less than 5 pixels wide since the data was gridded into 1 m cells, making it hard for them to survive the Gaussian low pass filtering and morphological operations needed to finalize the segmentation. This phenomenon can be seen up-close in Figure 55, where five buildings within Grand Goave went undetected as a result of their small size. Though small buildings were undetected in every scene,

Grand Goave had the largest number of small buildings out of any of the validation sites, which helped to explain its poor detection rate.

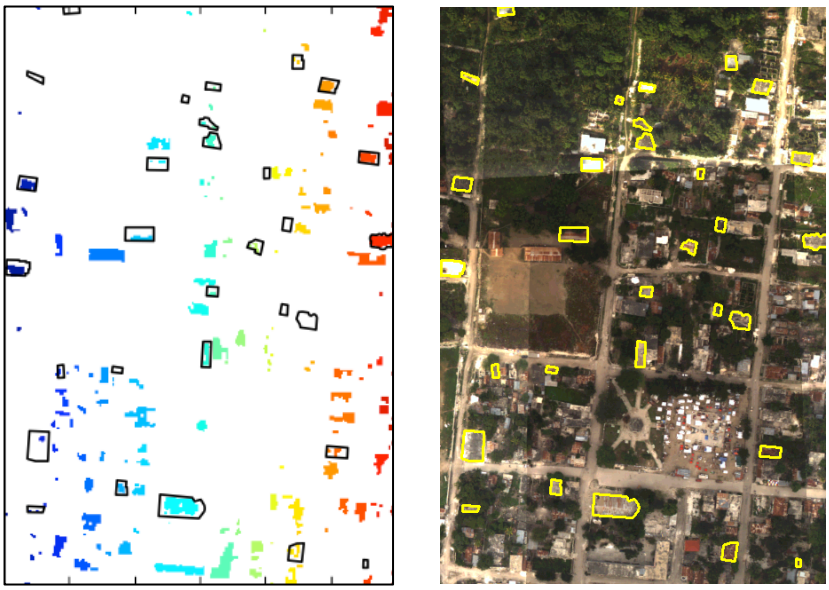


Figure 54. Truth building outlines for the Grand Goave validation site are overlaid on the output building map (*left*) and WASP imagery (*right*). Notice that many of the small building regions, typically those less than 5 m x 5 m in size, went undetected by the building segmentation routine.

Figure 55. Zoom-in area of Grand Goave showing five small buildings that went undetected by the building segmentation algorithm. The buildings were roughly 4.5 m wide, too small to survive the filtering and morphological operations that were used as part of the segmentation.

The final issue worth mentioning is that some of the GEO-CAN truth buildings that were used to assess segmentation performance, were not actual buildings. The result from the Darbonne site, where this issue was most commonly seen, is shown in Figure 56. Many of the large building outlines that did not encompass a segment from the

building map, and therefore were considered “undetected”, appeared to be plots of bare soil and piles of stone upon second review. In addition, some livestock corrals looked like destroyed buildings.

Examples of these questionable areas are shown in Figure 57: the blue building points representing GEO-CAN assessments did not all correspond to actual buildings, or even leveled buildings. These “buildings” were difficult to discern from the WASP imagery, but multi-temporal analysis of high resolution Google Earth imagery showed that at least two of the three unknown areas were in fact not buildings. In Darbonne, eight of these areas were incorrectly included in the building truth, since they appeared to be flattened buildings and were assigned a damage grade by GEO-CAN interpreters. These “buildings” went undetected, correctly, by the building segmentation algorithm. This issue, combined with the fact that there were a lot of small buildings in Darbonne, caused the validation site to have a detection rate of 66.7%, the worst of all the scenes. However, it is evident that the algorithm workflow arguably performed much better than the validation results would suggest in such cases.

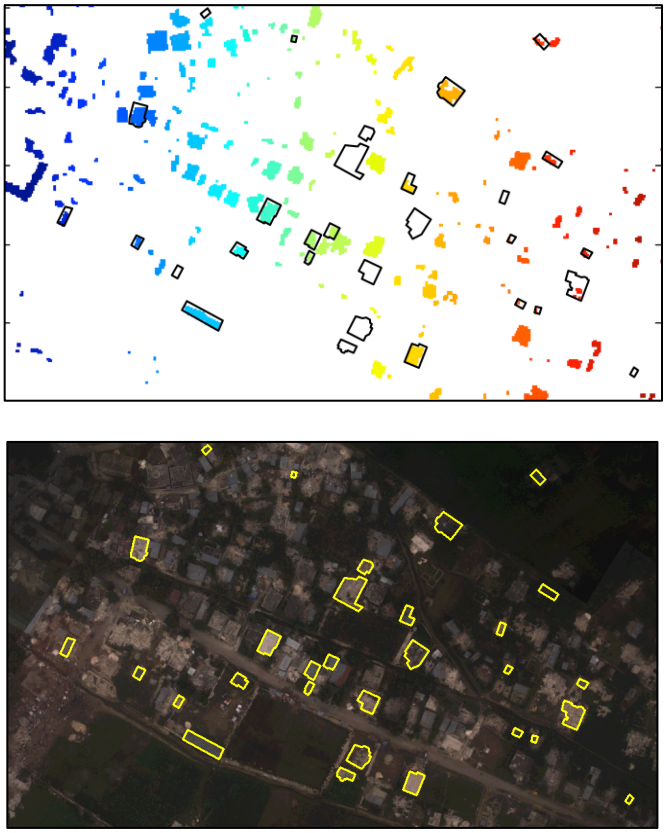


Figure 56. Truth building outlines for the Darbonne validation site are overlaid on the output building map (*top*) and WASP imagery (*bottom*). Ten of the 30 truth buildings went undetected in this scene, primarily due to small building size and questionable truth data.

Figure 57. Zoom-in area of the Darbonne site showing three questionable building regions that were all given GEO-CAN damage assessments. The two areas indicated by yellow arrows were incorrectly included in the truth data set. Google Earth imagery (*right*) was useful in discerning actual buildings from bare earth and livestock corrals.

Overall, the building segmentation algorithm was least effective in areas with a high concentration of buildings and when buildings were generally less than 5 m x 5 m in size. It performed well in areas where buildings were spread out and the algorithm could effectively remove vegetation from the scene. Buildings that were partially covered by trees were usually found, as long as some building structure was exposed. Unlike building density per area, the slope of the terrain did not seem to have an effect on building detection. The height models for the steep sites did not appear any different than those for the flat sites, which implies that the topographical effect was removed successfully. This was attributed to the accurate LiDAR ground point classification that was provided by Kucera International, the data vendor.

5.3 Building Damage Detection Performance

The damage detection algorithm classified each truth building segment as either undamaged (Grade 1) or damaged (Grades 3-5). The performance of the algorithm was evaluated by comparing the classification result to the actual damage rating assigned by GEO-CAN or EEFIT, where the latter was available. Since the building segmentation algorithm did not provide a building mask that was representative of all the buildings in the scenes, the truth building outlines were used as input to assess the damage detection routine. For each validation site, the building outline shapefile was imported into the tool and used to mask the building height model. This resulted in a set of building points, all referenced to ground, for each of the 30 truth building regions. The autonomous algorithm was then run iteratively on each of the building point clouds, using a 2 m x 2 m tile size for the normal angle and variance calculations.

Once a damage percentage was assigned to each of the building point clouds, a threshold had to be smartly chosen in order to discern between undamaged and damaged buildings. This was accomplished using the Palace and Darbonne scenes as test data. For both of the sites, the overall accuracy, omission, and commission errors were calculated for every percentage point from 1-100 (Congalton and Green, 1993). It was determined that a 51% threshold resulted in the best overall accuracy and omission error combination for both sites. This threshold was then used on the damage percentages generated for the rest of the validation sites.

A summary of the results for each of the validation sites is presented in Table 4. The table lists the overall damage detection accuracy for each site, as well as the kappa coefficient. Kappa, an estimate of overall accuracy, is used to determine how much better the classification results are than chance. In this case, it measured the agreement between the damage detection algorithm and the GEO-CAN/EEFIT truth classification. Kappa coefficients of one indicate that the classifiers are in complete agreement, where coefficients of zero or less indicate that the agreement is no better than chance (Schott, 2007). The confusion matrices, overall percent damage maps, and initial damage assessment maps for all of the sites are included in Appendix A. An example confusion matrix for the Palace validation site is shown in Table 5.

Based on overall accuracy, the damage detection algorithm performed the best on the Grand Goave validation site, with the Palace scene a close runner-up. The number of truth damaged buildings classified as undamaged buildings was also extremely low in both cases, i.e., one and zero, respectively. This was arguably as important as overall accuracy for this research, since missing or under-classifying heavy damage could have a

disastrous effect during emergency response. The algorithm performed the worst on the Turgeau scene, while the statistical output suggests it was no better than chance. Nine out of the 30 truth buildings in the site were misclassified as undamaged.

Table 4. Summary of results from running the autonomous damage detection routine on the 30 truth building point clouds from each of the six validation sites. The overall classification accuracy and the kappa coefficient are listed for each site.

Validation Site	Overall Accuracy (%)	Kappa Coefficient (%)
Palace	73.33	50
Darbonne	70	40
Grand Goave	76.67	25.53
Léogâne	66.67	30.56
Turgeau	50	-1.81
Riviere Froide	73.33	44.95

Sparse/Flat Sparse/Steep
Dense/Flat Dense/Steep

Table 5. Confusion matrix assessing the accuracy of the damage detection algorithm on the Palace validation site. The assessment, performed on the 30 truth building point clouds, provides the number of classified and misclassified buildings in terms of both buildings and percentages.

	Reference Data			
Building Damage Grade	1	3 - 5	Row Total	User's Accuracy
1	12	0	12	100.00%
3 - 5	8	10	18	55.56%
Column Total	20	10	30	
Producer's Accuracy	60.00%	100.00%		
Overall Accuracy	73.33%			
Kappa Coefficient	50.00%			

Several observations were made while analyzing the results from the damage detection algorithm and examining specific cases where the classification succeeded and failed. First, the effectiveness of the damage detection algorithm was directly related to

the quality of the input LiDAR point cloud. Even though the building point clouds were derived from the truth building shapefile, registration errors existed between the WASP imagery used to trace the outlines and the LiDAR height model. This caused some non-building points, or points from adjacent building regions, to be included in the building point clouds. An example of this phenomenon is show in Figure 58 using a building from the Grand Goave site. In this case, the added extra points created damaged normal vectors along the edges of the building, and the building was assigned a damage percentage of 36%. This did not exceed the threshold of 51%, however, so the algorithm correctly classified the building as undamaged.

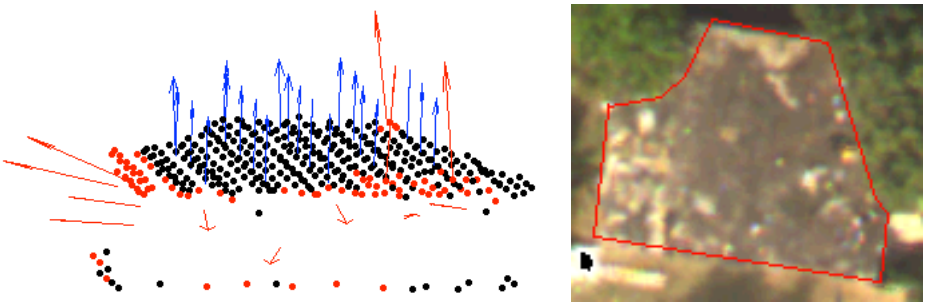


Figure 58. Some ground points are included in the building point cloud due to small geometric registration errors between the LiDAR height model and the WASP imagery, used to trace the building outline. The extra ground points affected the overall damage percentage, but not enough in this case for the algorithm to incorrectly classify the building as damaged.

Similarly, the building outlines that were traced did not always represent the actual building borders. In a few situations, the particular WASP image used to outline the buildings exhibited poor contrast and the resolution was not fine enough to discern building edges. This is evident in Figure 59, where the building outline exceeded the actual building edge. A number of tree points, ground points, and adjacent building points were added to the building point cloud in this case. These points caused the

algorithm to calculate a damage percentage of 63%, enough to misclassify the building as damaged.

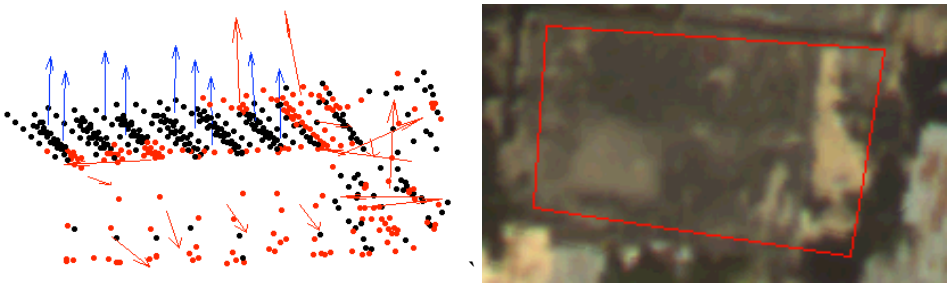


Figure 59. Building located in the Riviere Froide validation site whose outline was not traced accurately. This caused many tree points, ground points, and points from adjacent buildings to be included in the building point cloud. As a result, the algorithm detected enough damage to misclassify the building as Damage Grade 3-5.

These segmentation issues were evident in every validation site and contributed to most of the over-classification of damage for undamaged buildings. In five out of the six sites, six or more truth Grade 1 buildings were misclassified as damaged, which greatly impacted the overall accuracy. If the building point clouds were better representative of the actual buildings, this over-classification of damage would not be as significant and the accuracy would increase. However, it is promising in a theoretical sense that the algorithm flagged the extra points as damaged and not representative of the dominant roof planes. This indicated the algorithm was performing as intended.

The second observation made during the analysis of the results was that the algorithm did not perform well at roof joints, i.e., where two roof planes intersect. This was due to the fact that the damage detection routine tiled the point cloud, which resulted in some of the tiles spanning the roof joint. Roof joint points affect the plane that is best fit, which in turn causes the normal vector to be different than the normal vectors of the dominant roof planes. An example of this is shown in Figure 60, using a building from

the Palace validation site. Enough damage was detected along all the roof joints to misclassify the building as damaged.

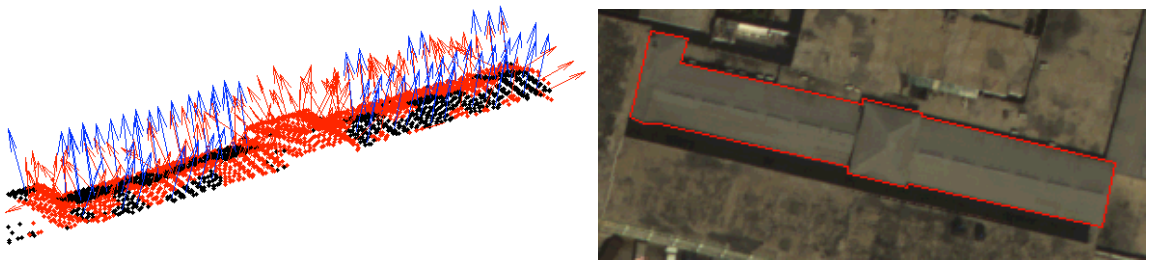


Figure 60. The damage detection routine incorrectly identified damage along all the roof joints due to the way the points were tiled. The truth undamaged building was determined to be 66% damaged (classified into Damage Grade 3-5).

It was also evident that if a small percentage of a building was damaged, yet the majority of the building was undamaged, the damage detection algorithm would classify the building as undamaged. This is due to the way the normal angle damage percentage metric was computed. Recall that the percent damage was defined by the number of damaged normal vectors (red) divided by the total number of normal vectors (red and blue). Figure 61 illustrates a case where the damage detection routine performs as intended for detecting damage, but the building is intact enough that it is only determined to be 42% damaged. As a result, the building was classified as undamaged. The GEO-CAN assessment, however, rated the building as Damage Grade 4.

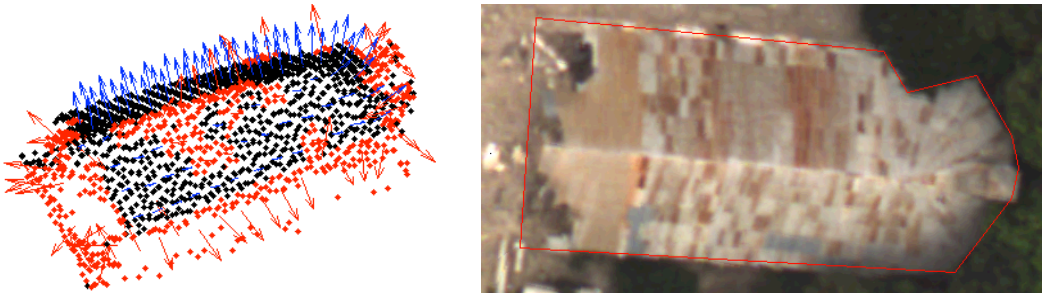


Figure 61. The major damage in this building from the Grand Goave validation site was detected, but it was not enough to classify the building as damaged. This was the only building in the Grand Goave site that was under-classified.

The damage detection algorithm furthermore was less effective in cases where buildings had small roof planes in addition to larger, dominant planes. This is illustrated in Figure 62, using a truth building from the Palace validation site. The routine correctly identified the two dominant roof planes, but classified all the points on the smaller plane as damaged. This issue is inherent to the algorithm, since it uses a threshold to determine the number of parallel normal vectors required to call a set of tiles a plane. The threshold value is based on a percentage of the total number of normal vectors generated for the building. In this case, there were simply not enough parallel normal vectors from the small plane, so the algorithm did not detect the plane and therefore classified all the points as damaged. This created enough damaged points to misclassify the entire building.

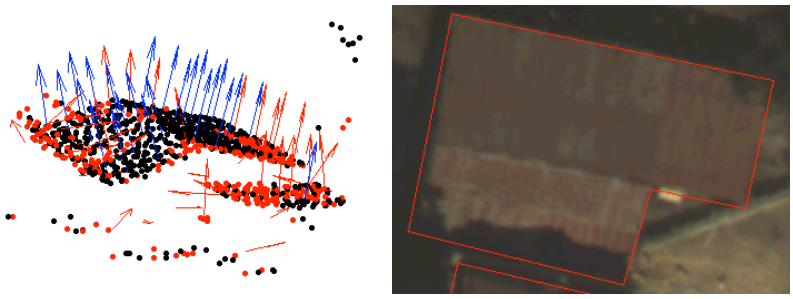


Figure 62. The damage detection routine did not recognize the smaller building plane and therefore classified all the points on that plane as damaged. The entire building was assessed to be 59% damaged and ended up being misclassified as Damage Grade 3-5. The building was rated Damage Grade 1 by the GEO-CAN effort.

As was the case with the building segmentation results, the performance of the damage detection algorithm depended greatly on the accuracy of the GEO-CAN and EEFIT assessments. This was very challenging, especially in the case of GEO-CAN, where assessments were based on hundreds of volunteers' individual interpretations of damage. It is hard to be critical of the GEO-CAN effort, since it was outstanding for the

amount of area it covered and the time it took to complete, however imperfections did surface when the buildings were closely examined.

In Léogâne, for example, 84 out of 295 GEO-CAN assessments were rated Damage Grade 3. For many of these "moderately damaged" buildings, the point cloud and high resolution imagery indicated that the buildings were in fact intact. This is shown in Figure 63, where the damage detection algorithm classified the building as undamaged, a finding that was confirmed by reviewing WASP and Google Earth imagery. This was the case for many of the buildings "incorrectly" classified as undamaged in Léogâne and was a major contributor to the poor accuracy achieved for the site. In other validation sites, where different individuals were interpreting results, the same buildings would have most likely been assessed as undamaged.

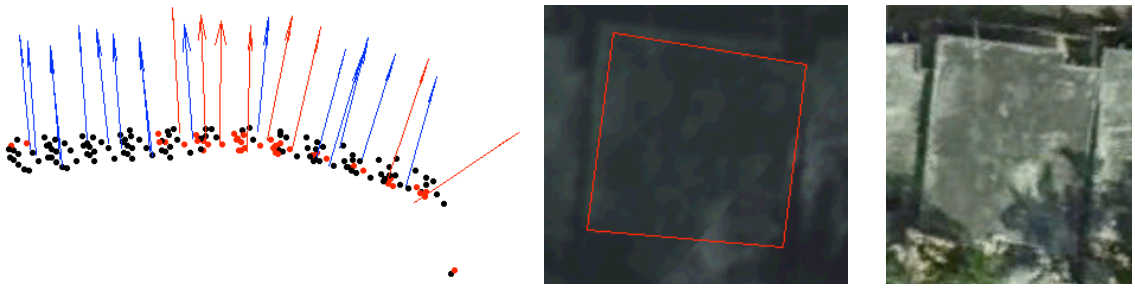


Figure 63. Léogâne building that was rated Damage Grade 3 by GEO-CAN, but was classified as undamaged by the damage detection routine. The LiDAR building point cloud (*left*), WASP image (*middle*), and Google Earth image (*right*) appear to confirm the results of the routine.

Much of the confusion and variability in the building assessments was based on the way the earthquake damage manifested itself in Haiti. According to the USGS/EERI Advance Reconnaissance Team Report, buildings in many cases were not designed and constructed to resist strong ground motions. Structures with concrete roofs and slabs were the most vulnerable, as the heavy concrete walls were often not reinforced and the concrete and mortar were of poor quality (Eberhard *et al*., 2010). When these buildings

collapsed as a result of the earthquake, the concrete slab roof often shifted but remained relatively intact, making the damage challenging to remotely sense.

Figure 64 shows an example of a partially collapsed building within the Turgeau validation site, which appeared to be undamaged based on assessments using the LiDAR and WASP data. The damage detection routine assigned a damage percentage of 29% to the building and thus classified it as Damage Grade 1. A look at the surrounding rubble, however, suggests that the building is likely collapsed. A similar example is shown in Figure 65, where the LiDAR assessment indicated that the building was undamaged, but in reality the building was partially collapsed. Note that the ground-based EEFIT assessment was used to determine the truth level of damage for this building. In both of these cases, the damage detection routine could not differentiate between an undamaged concrete roof slab and one that had shifted or collapsed, but remained relatively intact. This is a shortcoming of the algorithm, as it contributed to the under-classification of heavy damage.

Figure 64. Partially collapsed building in the Turgeau validation site that was classified as Damage Grade 1 by the damage detection routine. The concrete roof slab remained intact, which caused the algorithm to assume it was undamaged.

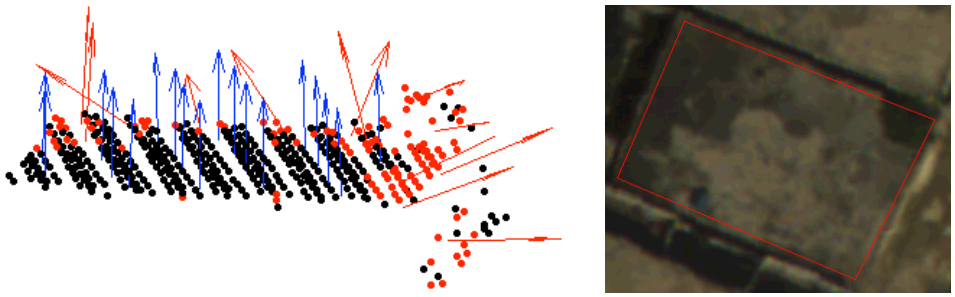


Figure 65. Building in the Turgeau validation site that appeared undamaged and was classified as Damage Grade 1 by the damage detection algorithm. Based on ground observations, EEFIT assessed the building at Damage Grade 3. This level and manifestation of damage is difficult to sense from a remote standpoint.

The accuracy of the results was heavily dependent on the threshold used to distinguish between undamaged and damaged buildings. There were several situations in which buildings with damage percentages right around the threshold were misclassified. Figure 66, for example, shows a building from the Riviere Froide validation site that had a damage percentage of 50%. This was just below the 51% threshold and was therefore classified as undamaged. The GEO-CAN rating for the building was Damage Grade 5, indicating the building was misclassified.

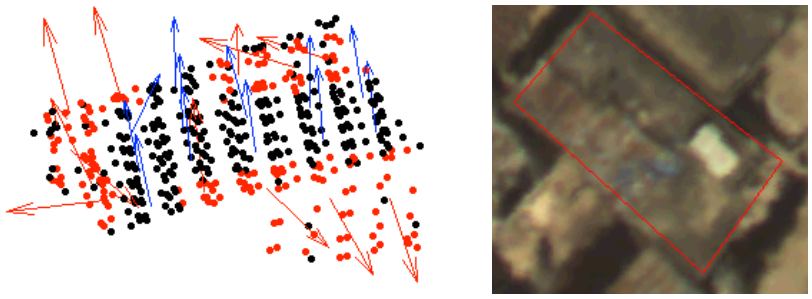


Figure 66. The damage detection routine calculated the percent damage for this building to be 50%, just shy of the 51% threshold for classifying a building as damaged.

It should be noted that the algorithm and associated workflow performed as expected, even given the selected case evidence of poor performance. As shown in most of the previous figures, the algorithm did a good job of detecting damage on flat roofs, or roofs with large dominant planes. An example of the algorithm correctly identifying

heavy damage is shown in Figure 67. The algorithm even was effective in the case of Haiti's Bicentennial Monument, shown in Figure 68. It could in fact be argued that much of the misclassification or under-classification of damage is due to the nature of the damage, environment, and data type. There is a high likelihood that the approach would perform better if a user were able to properly/exactly segment buildings in either sparser building environments or for higher density LiDAR point clouds, in the case of denser buildings scenes.

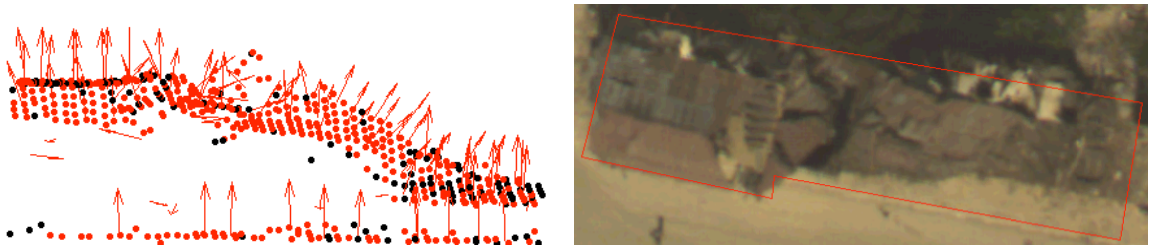


Figure 67. Collapsed Grade 5 building in the Palace validation site that was calculated to be 100% damaged.

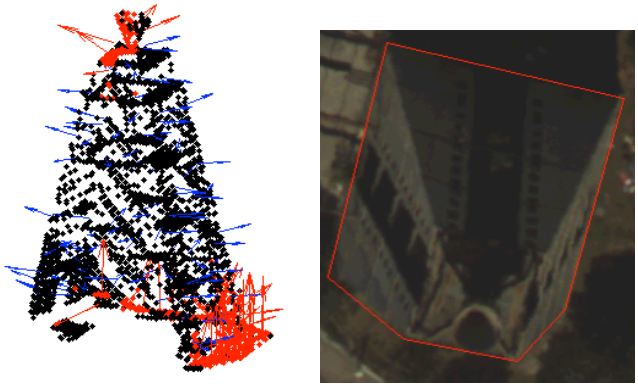


Figure 68. The damage detection algorithm performed well on the Bicentennial Monument, as it was correctly classified as undamaged.

5.4 Chapter Summary

In this chapter, the entire approach was demonstrated on a set of six validation sites carefully selected from the Haiti LiDAR data. The building segmentation algorithm was run on each site and the output building maps were compared to the building truth

sets. The algorithm detected a combined 85.6% of the truth buildings in all the sites, with a standard deviation of 15.3%. The damage detection algorithm classified each truth building as either undamaged (Grade 1) or damaged (Grades 3-5). The performance of the algorithm was evaluated by comparing the classification result to the actual damage rating assigned by GEO-CAN, or EEFIT. The combined overall classification accuracy for all six sites was 68.3%, with a standard deviation of 9.6%.

6 Conclusion

The use of LiDAR data to detect and quantify building damage following a natural disaster was investigated in this project. Using only LiDAR data collected after the Haiti earthquake, a set of processes was developed for mapping urban environments and assessing damage. The devised techniques were incorporated into a Matlab GUI, which guided the workflow and allowed for user interaction. The semi-autonomous tool ingested a discrete-return LiDAR point cloud of a post-disaster scene, and output a building damage map showing damaged and collapsed buildings.

Though the processes were not fully automated, the results were promising. Using six sites and 30 truth buildings selected for each site as validation data, the building segmentation algorithm detected an overall 85.6% of the buildings, with a standard deviation of 15.3%. The damage detection algorithm had an overall classification accuracy of 68.3%, with a standard deviation of 9.6%. The results indicate that medium density LiDAR can be used to quickly and accurately develop an initial damage assessment map immediately following a disaster.

Although this research produced promising results, there are many areas where the techniques could be improved. Building detection in Haiti proved challenging due to the varying building sizes, construction materials, and building densities per area. Since a "one size fits all" approach did not work, processes were developed to allow a user to select between operations and parameters that could be optimized for the particular scene being analyzed. This produced acceptable results, but required human interaction that took valuable time. If the building segmentation process were completely autonomous, the tool would become far more effective for disaster management.

One potential way to automate the building detection process is to use machine learning and pattern recognition techniques to classify the LiDAR point cloud and identify building points. A set of building characteristics could be determined, which numerically represents the building class. These features could then be combined into a feature vector to serve as training data for a statistical classifier.

Furthermore, the damage detection algorithm is based on measuring a point's deviation from a dominant roof plane in order to assess whether or not a building is damaged. Knowledge of all the planes that make up the roof of a building is essential in order to accurately assess damage using this method. Automatically detecting planes in a building point cloud is difficult and the process presented in this research could be refined to produce better results. A region-growing technique is recommended, where planar regions are grown from initial seed points that fall on the same plane. Region-growing could result in a more accurate set of roof planes and reduce the errors caused by complex roof lines and roof joints.

The damage detection algorithm could also be improved to better detect collapsed and partially collapsed building damage. In these cases, as long as the roof remains intact, the current method is of limited use for detecting damage. One way to improve the results would be to use the knowledge that concrete structures were often the most susceptible to damage, since the concrete slab roofs generally collapsed but remained intact. The algorithm thus could be extended to classify single-planed concrete roofs that are slanted as damaged, if intensity information could be used to distinguish concrete from other building materials. In addition, the algorithm could incorporate more contextual clues, such as rubble piled next to a building, into its damage assessment.

This research focused on using solely LiDAR data collected after a disaster to detect buildings and assess damage. Even with improvements to the algorithms, the results suggest that LiDAR technology can only go so far to accurately detect buildings and quantify structural damage. If the LiDAR data were fused with auxiliary data from other remote sensing or in situ sources, the results could be dramatically improved. High resolution imagery or ground plan information, for example, could be used to detect building edges or spectrally remove vegetation, which strengthens the building segmentation component. Imagery could also be used to drape the LiDAR point cloud, and provide better insight into building textures and surfaces. Multi-temporal LiDAR could also be used to more accurately perform damage detection. LiDAR datasets collected before and after a disaster could be compared and the changes could be quantified. In the case of Haiti, however, LiDAR data collected prior to the earthquake did not exist. The next potential area for improvement is evaluating the scale at which assessments are performed.

Depending on the application, damage assessment can be accomplished at various scales. This research produced damage results at the building level, which was consistent with the GEO-CAN and EEFIT efforts. Individual building damage assessments are useful to organizations like the World Bank, who typically plans and budgets for long-term disaster recovery. It might be useful, however, to extend the damage metrics for use at neighborhood scales. Instead of targeting specific buildings, damage assessment could be performed at the local scale, which would likely be of greater use to emergency responders immediately following a disaster. Such information would direct responders to general areas that require specialized attention, as opposed to an individual building.

It was evident, in conclusion, that the algorithm exhibited significant potential for detecting buildings and assessing the building-level damage. Although results were (i) skewed by extenuating factors, e.g., imperfect validation data from a worthwhile crowd-sourcing initiative (GEO-CAN), inclusion of non-building LiDAR returns, etc., and (ii) impacted by a very diverse environment, e.g., varying building types, sizes, and densities, the overall detection and classification accuracies were encouraging. It is likely that an increased LiDAR point density and algorithm refinements could lead to a much improved and potentially operational workflow. This approach could then evolve into a tool that is invaluable to disaster responders and decision makers during such high impact natural disasters.

References

Adams, B. J. (2006). The Emerging Role of Remote Sensing Technology in Emergency Management. In C. Taylor, & E. VanMarcke, *Infrastructure Risk Management Processes: Natural, Accidental, and Deliberate Hazards* (pp. 95-117). Reston, VA: American Society of Civil Engineers.

Alexander, C., Smith-Voysey, S., Jarvis, C., and Tansey, K. Integrating building footprints and LiDAR elevation data to classify roof structures and visualise buildings. *Computers, Environment and Urban Systems, 33*, 285–292.

Argall, P. and Sica, R. (2003). Lidar (Laser Radar). In T. G. Brown, K. Creath, H. Kogelnik, M. Kriss, J. Schmit, & M. Weber*, The Optics Encyclopedia* (Vol. 2, pp. 1305-1322). Wiley VCH.

Axelsson, P. (2000). DEM Generation from laser scanner data using TIN adaptive models. *International Archives of Photogrammetry and Remote Sensing, XXXIII*, 85-92.

Baltsavias, E. P. (1999). A comparison between photogrammetry and laser scanning. *ISPRS Journal of Photogrammetry and Remote Sensing, 54*, 83-94.

Bevington, J., Adams, B., and Eguchi, R. (2010). GEO-CAN Debuts to Map Haiti Damage. *Imaging Notes, 25* (2), 26-30.

Brovelli, M., Cannata, M., and Longoni, U. (2002). Managing and processing LiDAR data within GRASS. *Open Source GIS - GRASS Users Conference 2002.* Trento.

Brunn, A. and Weidner, U. (1997). Extracting Buildings from Digital Surface Models. *International Archives of Photogrammetry and Remote Sensing, XXXII*, 27-34.

Charaniya, A., Manduchi, R., and Lodha, S. (2004). Supervised Parametric Classification of Aerial LiDAR Data. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops.*

Chen, Y., Su, W., Li, J., and Sun, Z. (2009). Hierarchical object oriented classification using very high resolution imagery and LiDAR data over urban areas. *Advances in Space Research, 43*, 1101-1110.

Coastal Change Hazards (USGSb). (22 June 2011). Retrieved 3 July 2011 from U.S. Geological Survey: http://coastal.er.usgs.gov/hurricanes/cch.php

Congalton, R. and Green, K. (1993). A practical look at the sources of confusion in error matrix generation. *Photogrammetric Engineering and Remote Sensing, 59*, 641-644.

Davidson, S. and Smith, D. (January 2011). Order From Chaos: Haiti Flight Operations Coordination Center. *Air Land Sea Bulletin* , pp. 19-21.

Eberhard, M., Baldridge, S., Marshall, J., Mooney, W., and Rix, G. (2010). *The MW 7.0 Haiti earthquake of January 12, 2010; USGS/EERI Advance Reconnaissance Team Report: U.S. Geological Survey Open-File Report 2010-1048.*

EERI/ImageCat Workshop. (2010). *Remote Sensing and the GEO-CAN Community: Lessons from Haiti and Recommendations for the Future.*

Eguchi, R., Gill, S., Ghosh, S., Svekla, W., Adams, B., Evans, G., et al. (2010). The January 12, 2010 Haiti Earthquake: A Comprehensive Damage Assessment Using Very High Resolution Aerial Imagery. *8th International Workshop on Remote Sensing for Disaster Management.* Tokyo.

Elberink, S. O. and Maas, H.-G. (2000). The Use of Anisotropic Height Texture Measures for the Segmentation of Airborne Laser Scanner Data. *International Archives of Photogrammetry and Remote Sensing, XXXIII*.

Elmqvist, M. (2002). Ground Surface Estimation from Airborne Laser Scanner Data Using Active Shape Models. *International Archives of Photogrammetry and Remote Sensing, XXXIV*, 114-118.

Faulring, J. W., McKeown, D. M., van Aardt, J., Casterline, M. V., Bartlett, B. D., and Raqueno, N. (2011). Supporting relief efforts of the 2010 Haitian earthquake using an airborne multimodal remote sensing platform. In S. S. Shen, & P. E. Lewis (Ed.), *Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery XVII. 8048.* Orlando: SPIE.

Forlani, G., Nardinocchi, C., Scaioni, M., and Zingaretti, P. (2006). Complete classification of raw LiDAR data and 3D reconstruction of buildings. *Pattern Analysis and Applications, 8*, 357-374.

Gonzalez, R., Woods, R., and Eddins, S. (2009). *Digital Image Processing Using Matlab, Second Edition.* Gatesmark Publishing.

Haala, N., Brenner, C., and Anders, K.-H. (1998). 3D Urban GIS From Laser Altimeter and 2D Map Data. *International Archives of Photogrammetry and Remote Sensing*, *XXXII*, pp. 339-346.

Haiti Earthquake - Fact Sheet #66 (USAID). (6 August 2010). Retrieved 28 June 2011 from USAID: http://www.usaid.gov/our_work/humanitarian_assistance/disaster_assistance/countries/haiti/template/fs_sr/fy2010/haiti_eq_fs66_08-06-2010.pdf

Huyck, C. K., Adams, B. J., and Kehrlein, D. I. (2003). An evaluation of the role played by remote sensing technology following the World Trade Center attack. *Earthquake Engineering and Engineering Vibration, 2* (1), 159-168.

Jones, M. (January 2011). CRG Experience in Haiti. *Air Land Sea Bulletin*, pp. 4-9.

Kilian, J., Haala, N., and Englich, M. (1996). Capture and Evaluation of Airborne Laser Scanner Data. *International Archives of Photogrammetry and Remote Sensing, XXXI* (B3), 383-388.

Kraus, K. and Pfeifer, N. (1998). Determination of terrain models in wooded areas with airborne laser scanner data. *ISPRS Journal of Photogrammetry and Remote Sensing, 53*, 193-203.

Kwan, M.-P. and Ransberger, D. M. (2010). LiDAR assisted emergency response: Detection of transport network obstructions caused by major disasters. *Computers, Environment and Urban Systems, 34*, 179-188.

Lach, S. (2008). *Semi-Automated DIRSIG Scene Modeling from 3D LiDAR and Passive Imagery.* Dissertation, Rochester Institute of Technology, Chester F. Carlson Center for Imaging Science, Rochester.

Laefer, D. F. and Pradhan, A. R. (2006). Evacuation route selection based on tree-based hazards using LiDAR and GIS. *Journal of Transportation Engineering, 132* (4), 312-320.

Lindenberger, J. (1993). *Laser-Profilmessungen zur topographischen Geländeaufnahme.* Dissertation, Deutsche Geodätische Kommission, Munich.

Lohani, B. (18 August 2010). *Topographic LiDAR*. Retrieved 1 July 2011 from Airborne Altimetric LiDAR: Principle, Data Collection, Processing and Applications: http://home.iitk.ac.in/~blohani/LiDAR_Tutorial/Airborne_AltimetricLidar_Tutorial.htm

Ma, R. (2005). DEM Generation and Building Detection from LiDAR Data. *Photogrammetric Engineering and Remote Sensing, 71* (7), 847-854.

Magnitude 7.0 - Haiti Region (USGSa). (18 June 2011). Retrieved 28 June 2011 from U.S. Geological Survey: http://earthquake.usgs.gov/earthquakes/eqinthenews/2010/us2010rja6/#summary

Mandlburger, G., Hauer, C., Höfle, B., Habersack, H., and Pfeifer, N. (2008). Optimisation of LiDAR derived terrain models for river flow modelling. *Hydrology and Earth System Sciences*.

Messinger, D. W., van Aardt, J., McKeown, D., Casterline, M., Faulring, J., Raqueno, N., *et al.* (2010). High-resolution and LiDAR imaging support to the Haiti earthquake relief effort. In S. S. Shen, & P. E. Lewis (Ed.), *Imaging Spectrometry XV. 7812.* Orlando: SPIE.

Morgan, M. and Habib, A. (2002). Interpolation of LiDAR Data and Automatic Building Extraction. *ACSM-ASPRS 2002 Annual Conference.* Washington D.C.

Morgan, M. and Tempfli, K. (2000). Automatic Building Extraction from Airborne Laser Scanning Data. *International Archives of Photogrammetry and Remote Sensing, 33* (B3), 616-623.

OpenTopography. Retrieved 6 July 2011 from Point Cloud Datasets Statistics: http://www.opentopography.org/index.php/about/datasetsmetrics

Pfeifer, N., Stadler, P., and Briese, C. (2001). Derivation of Digital Terrain Models in the SCOP++ Environment. *OEEPE Workshop on Airborne Laserscanning and Interferometric SAR for Digital Elevation Models.* Stockholm.

Rehor, M. (2007). Classification of building damages based on laser scanning data. *International Archives of Photogrammetry and Remote Sensing*, *XXXVI*, pp. 326-331. Espoo.

Rehor, M., Bähr, H.-P., Tarsha-Kurdi, F., Landes, T., and Grussenmeyer, P. (2008). Contribution of Two Plane Detection Algorithms to Recognition of Intact and Damaged Buildings in LiDAR Data. *The Photogrammetric Record, 23* (124), 441-456.

ReliefWeb. (26 January 2010). Retrieved 30 June 2011 from Damage Assessment and Field Medical Locations - Port Au Prince, Haiti: http://reliefweb.int/node/15615

Roth, R. and Thompson, J. (2008). Practical Application of Multiple Pulse in Air (MPiA) LiDAR in Large-Area Surveys. *The International Archives of the Photogrammetry,*

Remote Sensing and Spatial Information Sciences. XXXVII, pp. 183-188. Beijing: ISPRS.

Rottensteiner, F. and Briese, C. (2002). A New Method for Building Extraction in Urban Areas From High-Resolution LiDAR Data. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, XXXIV* (3A), 295-301.

Schott, J. R. (2007). *Remote Sensing: The Image Chain Approach.* New York, NY: Oxford University Press.

Schweier, C. and Markus, M. (2004). Assessment of the Search and Rescue Demand for Individual Buildings. *13th World Conference on Earthquake Engineering.* Vancouver.

Sithole, G. (2001). Filtering of Laser Altimetry Data Using a Slope Adaptive Filter. *International Archives of Photogrammetry and Remote Sensing, XXXIV*.

Sohn, G. and Dowman, I. (2007). Data fusion of high-resolution satellite imagery and LiDAR data for automatic building extraction. *ISPRS Journal of Photogrammetry and Remote Sensing, 62*, 43-63.

Tarsha-Kurdi, F., Landes, T., and Grussenmeyer, P. (2007). Hough-Transform and Extended RANSAC Algorithms for Automatic Detection of 3D Building Roof Planes from LiDAR Data. *International Archives of Photogrammetry and Remote Sensing, XXXVI*, 407-412.

Terrapoint. (2008). Retrieved 1 July 2011 from A White Paper on LiDAR Mapping: http://www.ambercore.com/files/TerrapointWhitePaper.pdf

TerraScan User's Guide. (3 October 2011). Retrieved 5 July 2011 from TerraSolid: http://www.terrasolid.fi/system/files/tscan_2.pdf

The World Factbook: Haiti. (14 June 2011). Retrieved 28 June 2011 from CIA: https://www.cia.gov/library/publications/the-world-factbook/geos/ha.html#

Vögtle, T. and Steinle, E. (2004). Detection and recognition of changes in building geometry derived from multitemporal laser scanning data. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Science, XXXV*, 428-433.

van den Broek, B., Kiefl, R., Riedlinger, T., Scholte, K., Granica, K., Gutjahr, K., et al. (2009). Chapter 16: Rapid Mapping and Damage Assessment. In B. Jasani, M.

Pesaresi, S. Schneiderbauer, & Z. G., *Remote Sensing from Space: Supporting International Peace and Security* (pp. 261-286). Springer.

Van Den Eeckhaut, M., Poesen, J., Verstraeten, G., Vanacker, V., Nyssen, J., Moeyersons, J., et al. (2007). Use of LiDAR-derived images for mapping old landslides under forest. *Earth Surface Processes and Landforms, 32*, 754-769.

Vosselman, G. (2009). Advanced Point Cloud Processing. *Proceedings of the Photogrammetric Week*, (pp. 137-145).

Vosselman, G. (2000). Slope based filtering of laser altimetry data. *International Archives of Photogrammetry and Remote Sensing, XXXIII*.

Vosselman, G. and Dijkman, S. (2001). 3D Building Model Reconstruction from Point Clouds and Ground Plans. *International Archives of Photogrammetry and Remote Sensing, XXXIV*.

Vu, T. T., Matsuoka, M., and Yamazaki, F. (2004). Employment of LiDAR for Disaster Assessment. *Proceedings of the 2nd International Workshop on Remote Sensing for Post-Disaster Response.* Newport Beach.

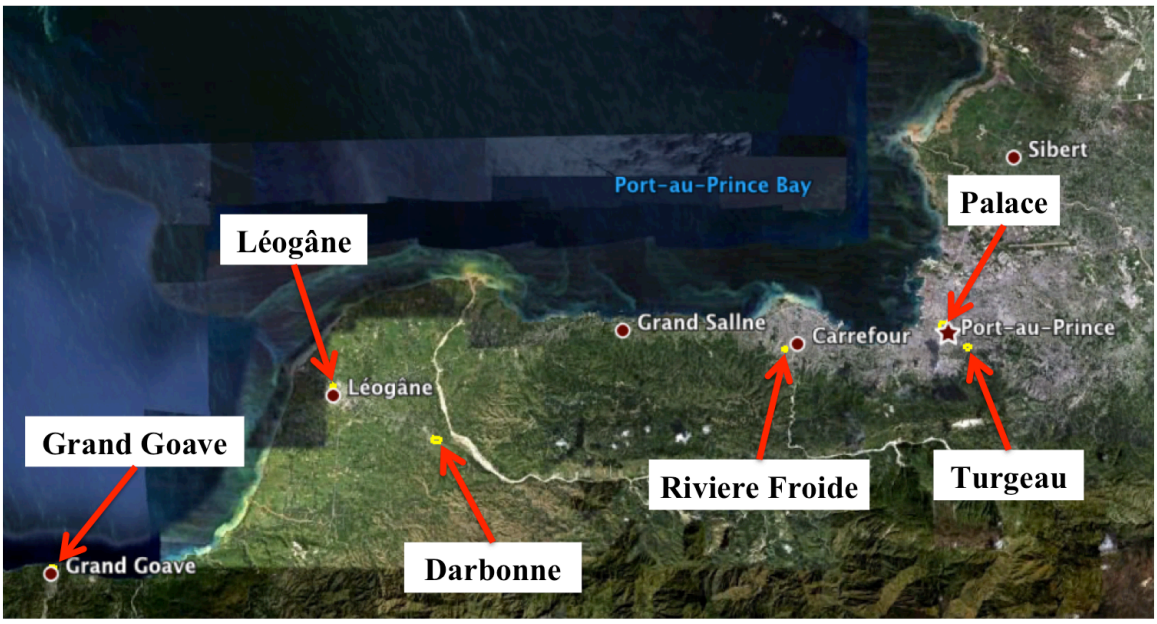
Vu, T., Yamazaki, F., and Matsuoka, M. (2009). Multi-scale solution for building extraction from LiDAR and image data. *International Journal of Applied Earth Observation and Geoinformation, 11*, 281-289.

Weidner, U. and Förstner, W. (1995). Towards automatic building extraction from high-resolution digital elevation models. *ISPRS Journal of Photogrammetry and Remote Sensing, 50* (4), 38-49.

Zhang, K., Chen, S., Whitman, D., Shyu, M., Yan, J., & Zhang, C. (2003). A Progressive Morphological Filter for Removing Nonground Measurements From Airborne LiDAR Data. *IEEE Transactions on Geoscience and Remote Sensing, 41* (4), 872-882.

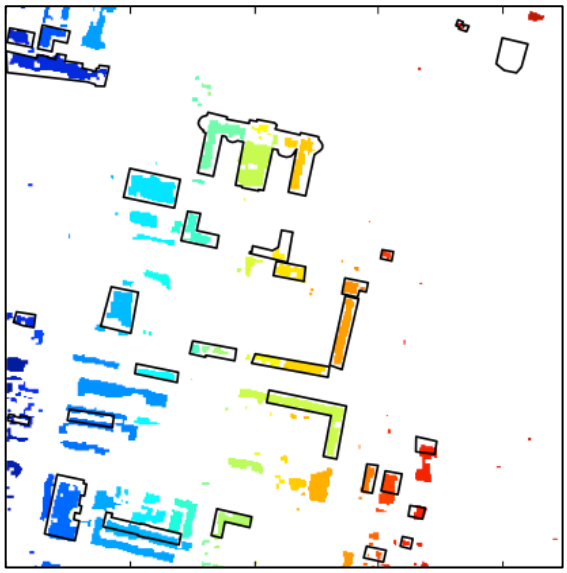
Appendix A

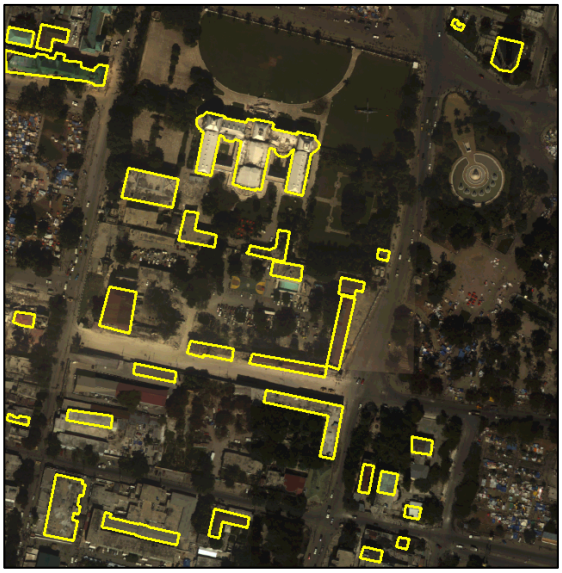
A.1 Validation Sites



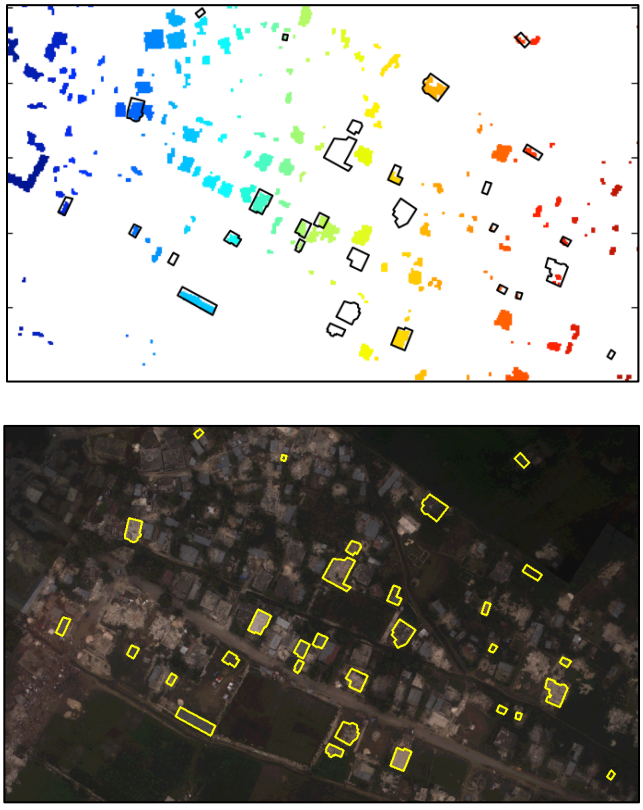
A.2 Building Segmentation Results – Output Building Maps

Palace

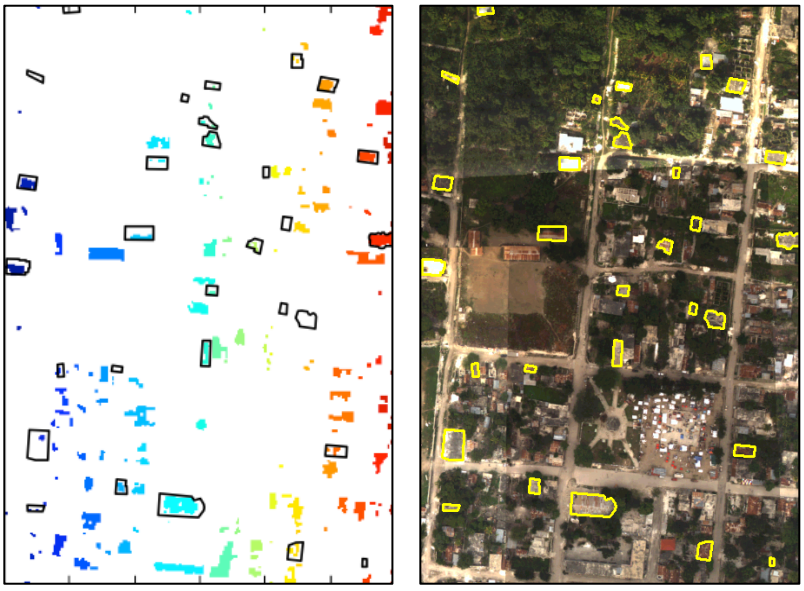




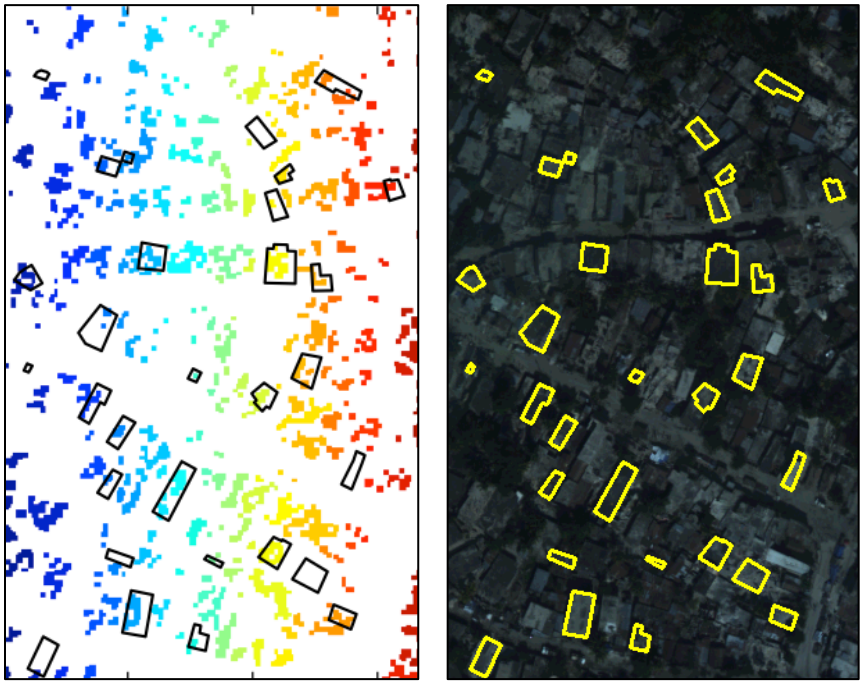
Darbonne



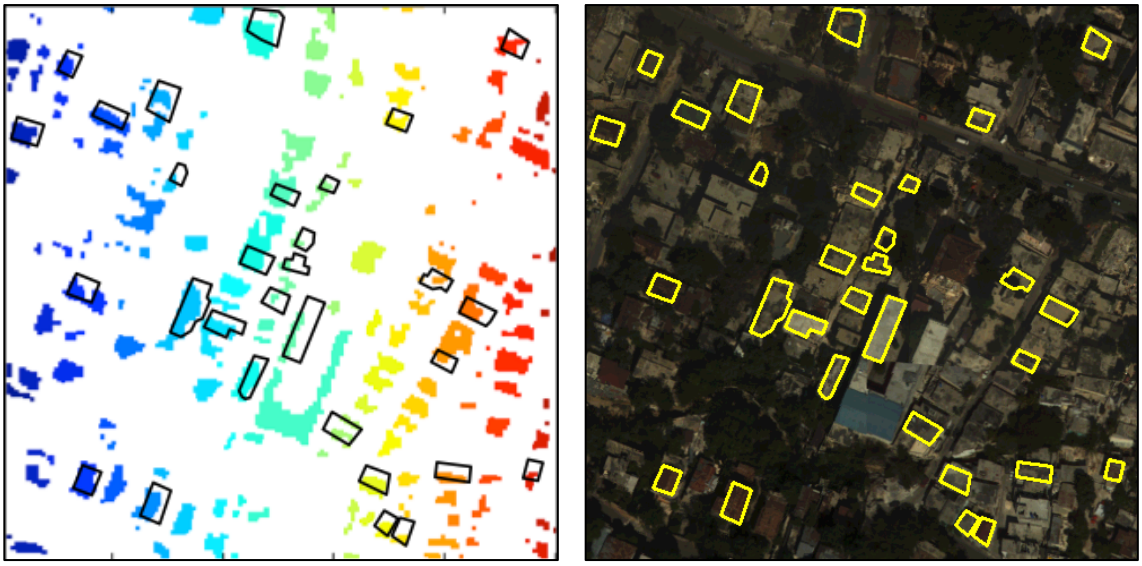
Grand Goave



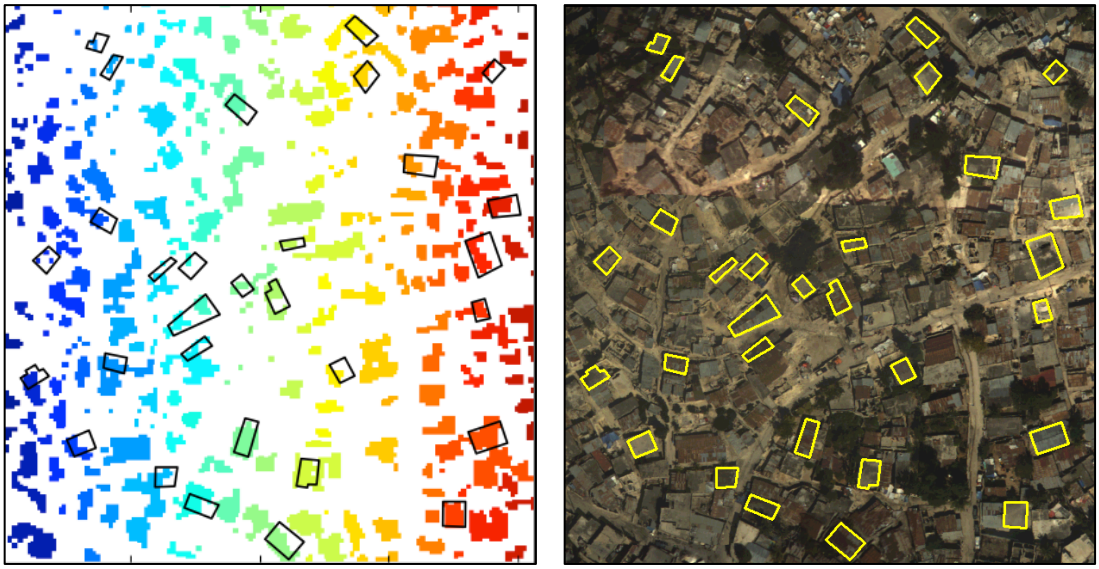
Léogâne



Turgeau

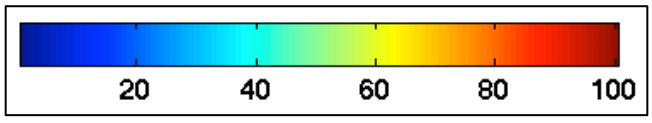


Riviere Froide



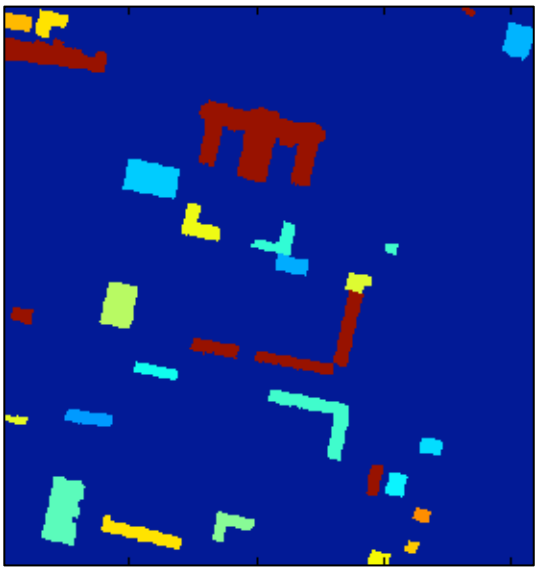
A.3 Damage Detection Results

The confusion matrices, overall percent damage maps, and initial damage assessment maps for all of the validation sites are included in this section. The building regions in the percent damage map are color-coded by percent damage, according to the colormap below. Buildings classified as undamaged (Grade 1) are shown in black and buildings classified as damaged (Grades 3-5) are colored red in the initial damage assessment map.



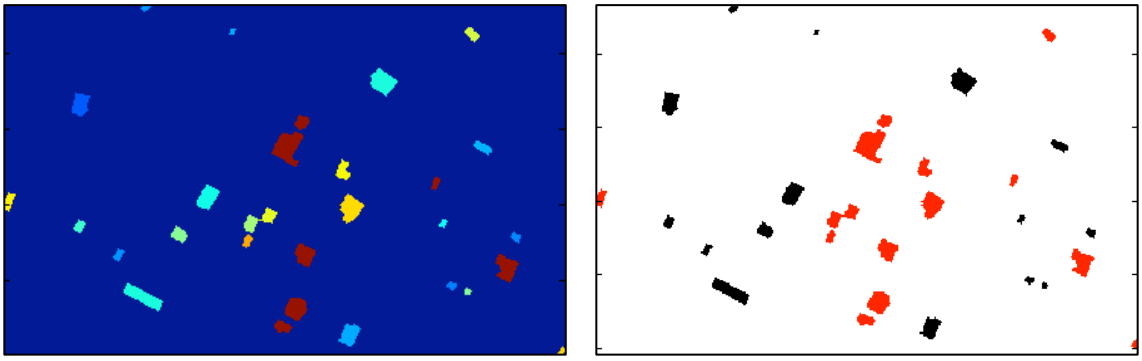
Palace

	Reference Data			
Building Damage Grade	1	3 - 5	Row Total	User's Accuracy
1	12	0	12	100.00%
3 - 5	8	10	18	55.56%
Column Total	20	10	30	
Producer's Accuracy	60.00%	100.00%		
Overall Accuracy	73.33%			
Kappa Coefficient	50.00%			



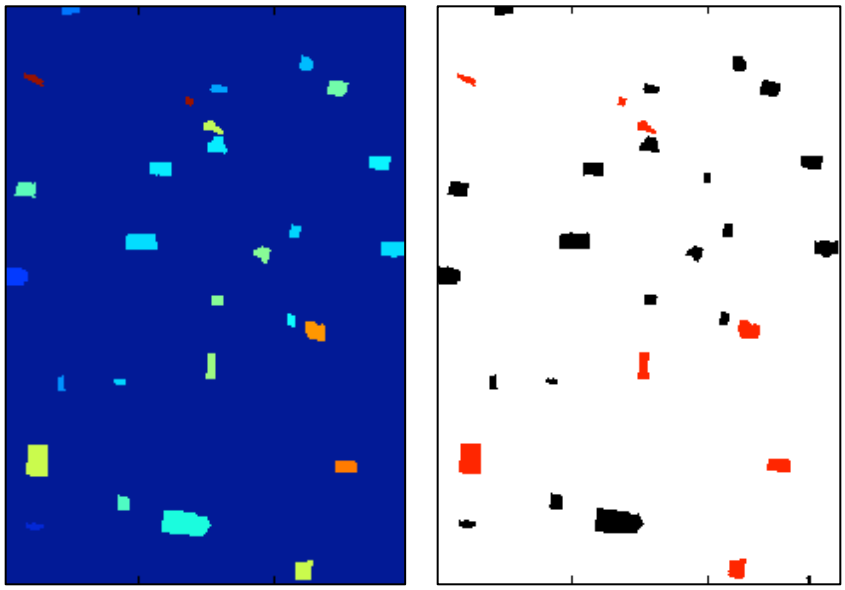
Darbonne

	Reference Data			
Building Damage Grade	1	3 - 5	Row Total	User's Accuracy
1	12	3	15	80.00%
3 - 5	6	9	15	60.00%
Column Total	18	12	30	
Producer's Accuracy	66.67%	75.00%		
Overall Accuracy	70.00%			
Kappa Coefficient	40.00%			



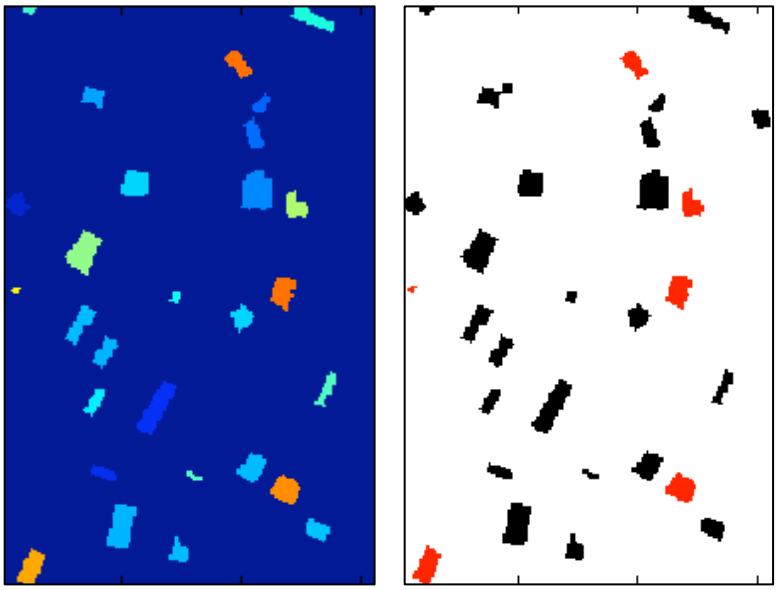
Grand Goave

	Reference Data			
Building Damage Grade	1	3 - 5	Row Total	User's Accuracy
1	21	1	22	95.45%
3 - 5	6	2	8	25.00%
Column Total	27	3	30	
Producer's Accuracy	77.78%	66.67%		
Overall Accuracy	76.67%			
Kappa Coefficient	25.53%			



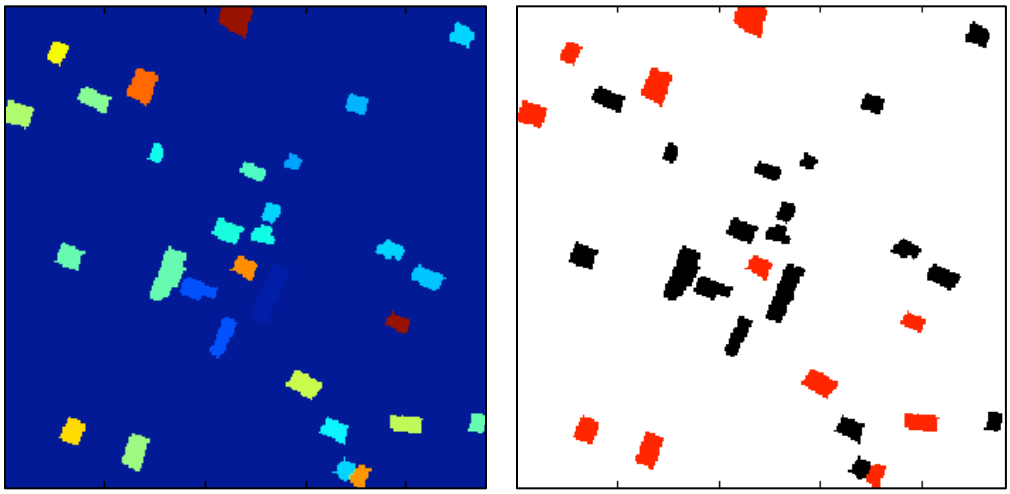
Léogâne

	Reference Data			
Building Damage Grade	1	3 - 5	Row Total	User's Accuracy
1	15	9	24	62.50%
3 - 5	1	5	6	83.33%
Column Total	16	14	30	
Producer's Accuracy	93.75%	35.71%		
Overall Accuracy	66.67%			
Kappa Coefficient	30.56%			



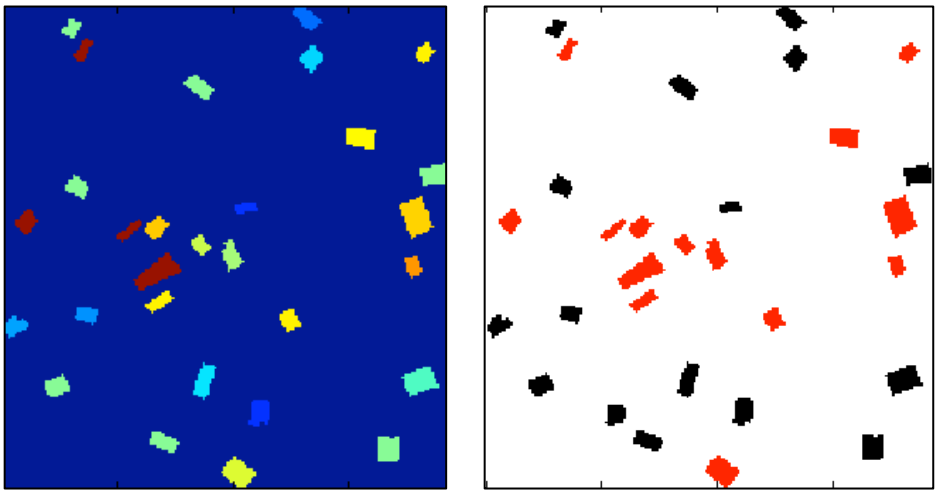
Turgeau

	Reference Data			
Building Damage Grade	1	3 - 5	Row Total	User's Accuracy
1	10	9	19	52.63%
3 - 5	6	5	11	45.45%
Column Total	16	14	30	
Producer's Accuracy	62.50%	35.71%		
Overall Accuracy	50.00%			
Kappa Coefficient	-1.81%			



Riviere Froide

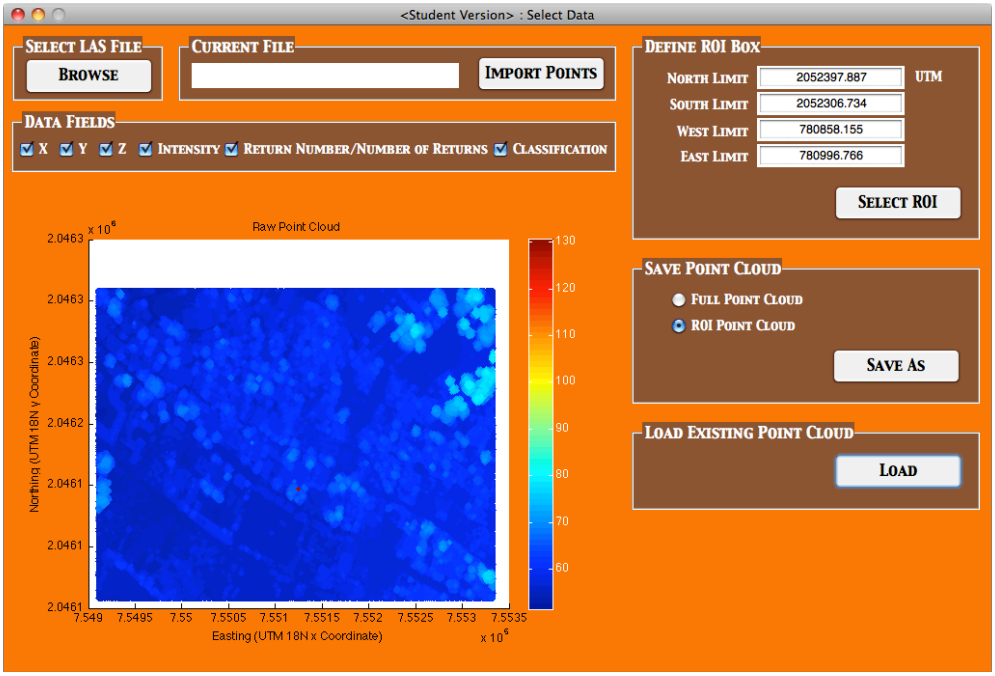
	Reference Data			
Building Damage Grade	1	3 - 5	Row Total	User's Accuracy
1	15	1	16	93.75%
3 - 5	7	7	14	50.00%
Column Total	22	8	30	
Producer's Accuracy	68.18%	87.50%		
Overall Accuracy	73.33%			
Kappa Coefficient	44.95%			



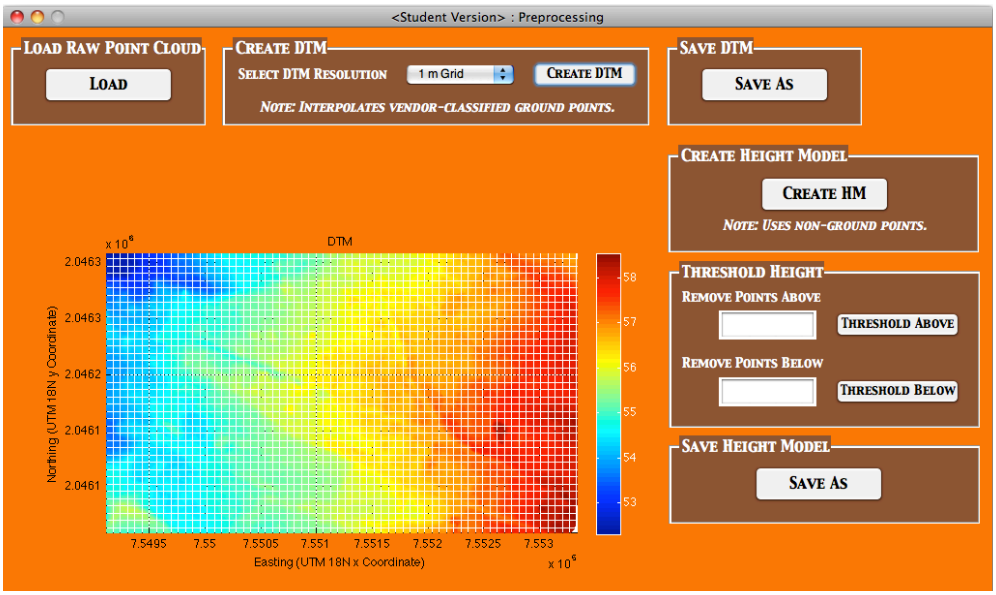
Appendix B

B.1 Matlab GUI Screenshots

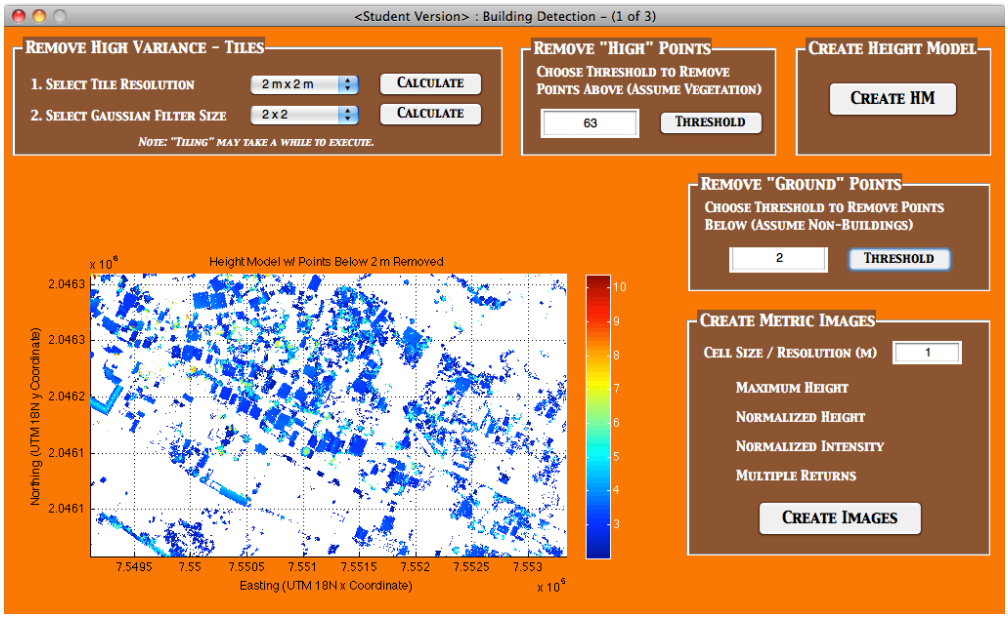
Data Selection – Reads in .las, .mat, or .txt point files and allows for ROI extraction



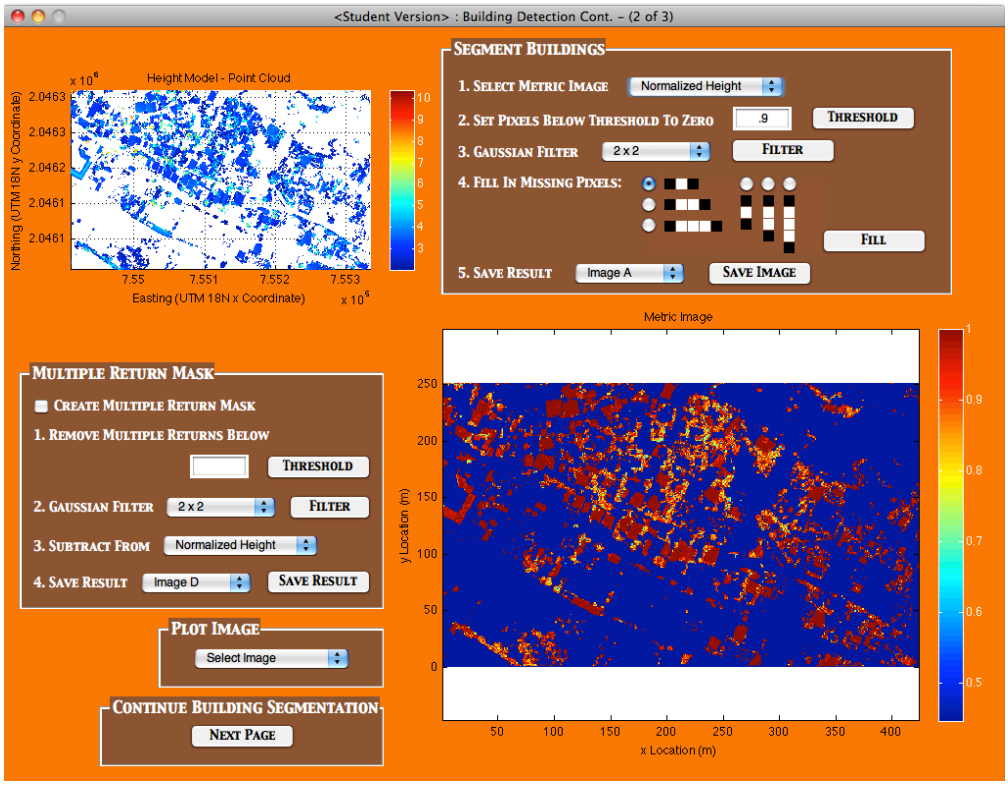
Preprocessing – Creates DTM and then subtracts from raw point cloud to produce referenced height model (normalized surface model)



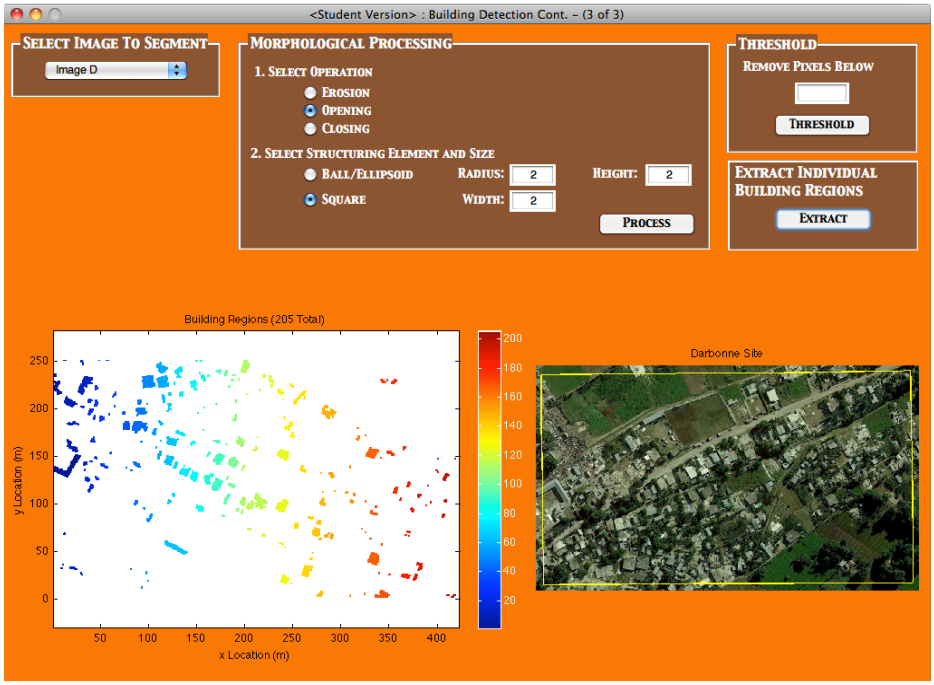
Building Detection (1 of 3) – Removes high variance (vegetation) points and creates initial building mask by removing points below 2 m from "thinned" height model



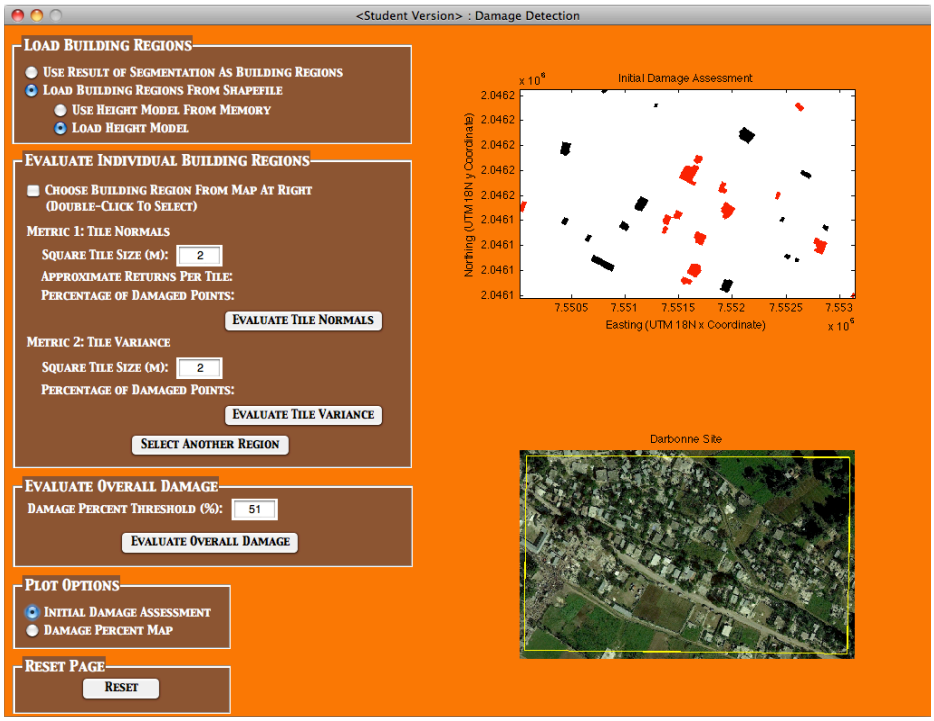
Building Detection (2 of 3) – Further defines building segments by thresholding and filtering metric images and masking out the multiple returns



Building Detection (3 of 3) – Finalizes and extracts individual building regions using morphological processing and connected components analysis



Damage Detection – Performs damage assessment using normal vector and height variance metrics to produce initial damage assessment and damage percent maps



B.2 Matlab Code

MainMenu_GUI.m

```
function varargout = MainMenu_GUI(varargin)
% MAINMENU_GUI M-file for MainMenu_GUI.fig
% MainMenu_GUI is the menu page for the operational tool and allows the
% user to navigate between the different applications.
%
% Rick Labiak
% Last Modified 11 August 2011

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @MainMenu_GUI_OpeningFcn, ...
    'gui_OutputFcn',  @MainMenu_GUI_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before MainMenu_GUI is made visible.
function MainMenu_GUI_OpeningFcn(hObject, eventdata, handles, varargin)

% Position window in screen
screen_size = get(0,'ScreenSize');
set(hObject,'Position',[0,screen_size(4)-364,480,364]);

% Load the background image into Matlab
backgroundImage = imread('MainFigureBkgrd.png');
% Select the axes
axes(handles.axes1);
% Place image onto the axes
image(backgroundImage);
% Remove the axis tick marks
axis off

% Choose default command line output for MainMenu_GUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = MainMenu_GUI_OutputFcn(hObject, eventdata,
```

```
handles)

% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on mouse press over MainMenu
function MainMenu_GUI_WindowButtonUpFcn(hObject, eventdata, handles)
% Get position of mouse
mouse_pos = get(handles.MainMenu_GUI,'CurrentPoint');
% Check which button the user pressed
if mouse_pos(1) >= 240 && mouse_pos(1) <= 457
    if mouse_pos(2) >= 277 && mouse_pos(2) <= 348
        %Open Select Data
        eval('SelectData')
    elseif mouse_pos(2) >= 191 && mouse_pos(2) <= 262
        %Preprocess
        eval('Preprocessing')
    elseif mouse_pos(2) >= 103 && mouse_pos(2) <= 174
        %Building Detection
        eval('BuildingDetect')
    elseif mouse_pos(2) >= 16 && mouse_pos(2) <= 87
        %Damage Detection
        eval('DamageDetectionFINAL')
    else
    end
else
end
```

SelectData.m

```
function varargout = SelectData(varargin)
% SELECTDATA M-file for SelectData.fig
% SelectData reads in a LAS, .mat, or .txt point file, and allows for
% ROI extraction.
%
% Rick Labiak
% Last Modified 11 August 2011

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @SelectData_OpeningFcn, ...
                   'gui_OutputFcn',  @SelectData_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
```

```
% End initialization code - DO NOT EDIT

% --- Executes just before SelectData is made visible.
function SelectData_OpeningFcn(hObject, eventdata, handles, varargin)
% Position window in screen
screen_size = get(0,'ScreenSize');
set(hObject,'Position',[screen_size(3)-1024,screen_size(4)/2-
335,1024,670]);

% Choose default command line output for SelectData
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = SelectData_OutputFcn(hObject, eventdata, handles)
% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on button press in x_checkbox.
function x_checkbox_Callback(hObject, eventdata, handles)

% --- Executes on button press in y_checkbox.
function y_checkbox_Callback(hObject, eventdata, handles)

% --- Executes on button press in z_checkbox.
function z_checkbox_Callback(hObject, eventdata, handles)

% --- Executes on button press in i_checkbox.
function i_checkbox_Callback(hObject, eventdata, handles)

% --- Executes on button press in r_checkbox.
function r_checkbox_Callback(hObject, eventdata, handles)

% --- Executes on button press in c_checkbox.
function c_checkbox_Callback(hObject, eventdata, handles)

% --- Executes on button press in open_pushbutton.
function open_pushbutton_Callback(hObject, eventdata, handles)
global my_handles
% Allows the user to interactively pick a .LAS file to open
[FileName,PathName,FilterIndex] = uigetfile('*.las','Select .las
file');
% Displays path and file name of selected file
fullpath = [PathName FileName];
set(handles.openfiletxt, 'String', fullpath);
handles.FileName = FileName;
handles.PathName = PathName;
my_handles.fullpath = fullpath;
guidata(hObject,handles);

% --- Executes on button press in importraw.
function importraw_Callback(hObject, eventdata, handles)
global my_handles
% Check toggle state of x_checkbox and save as GUI data
if (get(handles.x_checkbox,'Value') == get(handles.x_checkbox,'Max'))
```

```
    my_handles.xcheck = 1;
else
    my_handles.xcheck = 0;
end
guidata(handles.x_checkbox,handles);
% Check toggle state of y_checkbox and save as GUI data
if (get(handles.y_checkbox,'Value') == get(handles.y_checkbox,'Max'))
    my_handles.ycheck = 1;
else
    my_handles.ycheck = 0;
end
guidata(handles.y_checkbox,handles);
% Check toggle state of z_checkbox and save as GUI data
if (get(handles.z_checkbox,'Value') == get(handles.z_checkbox,'Max'))
    my_handles.zcheck = 1;
else
    my_handles.zcheck = 0;
end
guidata(handles.z_checkbox,handles);
% Check toggle state of i_checkbox and save as GUI data
if (get(handles.i_checkbox,'Value') == get(handles.i_checkbox,'Max'))
    my_handles.icheck = 1;
else
    my_handles.icheck = 0;
end
guidata(handles.i_checkbox,handles);
% Check toggle state of r_checkbox and save as GUI data
if (get(handles.r_checkbox,'Value') == get(handles.r_checkbox,'Max'))
    my_handles.rcheck = 1;
else
    my_handles.rcheck = 0;
end
guidata(handles.r_checkbox,handles);
% Check toggle state of c_checkbox and save as GUI data
if (get(handles.c_checkbox,'Value') == get(handles.c_checkbox,'Max'))
    my_handles.ccheck = 1;
else
    my_handles.ccheck = 0;
end
guidata(handles.c_checkbox,handles);

% Determine selected data fields and import data
my_handles.nFields = numfields(my_handles.xcheck, my_handles.ycheck,
my_handles.zcheck, my_handles.icheck, my_handles.rcheck,
my_handles.ccheck);
my_handles.fullcloud =
LASImport(my_handles.fullpath,my_handles.nFields);

% Plot raw point cloud using fscatter3 function developed by Felix
Morsdorf
axes(handles.axes1);
cla
fscatter3(my_handles.fullcloud(:,1),my_handles.fullcloud(:,2),my_handle
s.fullcloud(:,3),my_handles.fullcloud(:,3),jet)
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height ASL (m)'), title('Raw Point Cloud')
guidata(hObject,handles);
```

```
function north_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function north_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function south_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function south_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function east_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function east_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function west_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function west_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in selectROI.
function selectROI_Callback(hObject, eventdata, handles)
global my_handles
handles.NorthLimit = str2double(get(handles.north_edit,'String'));
handles.SouthLimit = str2double(get(handles.south_edit,'String'));
handles.WestLimit = str2double(get(handles.west_edit,'String'));
handles.EastLimit = str2double(get(handles.east_edit,'String'));

% Find ROI within point cloud
index = find(my_handles.fullcloud(:,1) < handles.EastLimit & ...
            my_handles.fullcloud(:,1) > handles.WestLimit  & ...
            my_handles.fullcloud(:,2) > handles.SouthLimit & ...
            my_handles.fullcloud(:,2) < handles.NorthLimit);
my_handles.roi = my_handles.fullcloud(index,:);

% Replot ROI
axes(handles.axes1);
cla
fscatter3(my_handles.roi(:,1),my_handles.roi(:,2),my_handles.roi(:,3),m
y_handles.roi(:,3),jet)
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height ASL (m)'), title('Raw Point Cloud')
guidata(hObject,handles);
```

```
% --- Executes on button press in savebutton.
function savebutton_Callback(hObject, eventdata, handles)
global my_handles
[FileName,PathName] = uiputfile({'*.mat';'*.txt'},'Save As');
entirepath = [PathName FileName];
[pathstr, name, ext] = fileparts(entirepath);
if strcmp(ext,'.txt')
    if (get(handles.fullcloudsave,'Value') ==
get(handles.fullcloudsave,'Max'))
        dlmwrite(entirepath,my_handles.fullcloud,'precision','%11.8g');
    else
        dlmwrite(entirepath,my_handles.roi,'precision','%11.8g');
    end
elseif strcmp(ext,'.mat')
    if (get(handles.fullcloudsave,'Value') ==
get(handles.fullcloudsave,'Max'))
        Full = my_handles.fullcloud;
        save(entirepath,'Full')
    else
        ROI = my_handles.roi;
        save(entirepath,'ROI')
    end
end

% --- Executes on button press in loaddata.
function loaddata_Callback(hObject, eventdata, handles)
global my_handles
[FileName,PathName] = uigetfile({'*.mat';'*.txt'},'Select File to
Load');
entirepath = [PathName FileName];
my_handles.fullpath = entirepath;
ROI = importdata(entirepath);
% Plot ROI
axes(handles.axes1);
cla
fscatter3(ROI(:,1),ROI(:,2),ROI(:,3),ROI(:,3),jet);
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height ASL (m)'), title('Raw Point Cloud')
my_handles.roi = ROI;
guidata(hObject,handles);
```

numfields.m

```
function [nFields] = numfields(xcheck, ycheck, zcheck, icheck, rcheck,
ccheck)
% numfields determines the number of data fields to be read in from the
% LAS file for each point record (based on user selection of
% parameters).
%
% Rick Labiak
% Last Modified 11 August 2011
% Based on LASRead Function by Cici Alexander (September 2008)

fieldmap = zeros(1,6);
```

```
fieldmap = [xcheck ycheck zcheck icheck rcheck ccheck];
if fieldmap == [0 0 0 0 0 0]
    % Error message for not enough parameters
    errordlg('X, Y, and Z coordinates need to be selected at a
minimum!','Error Dialog Box');
elseif fieldmap == [1 0 0 0 0 0]
    % Error message for not enough parameters
    errordlg('X, Y, and Z coordinates need to be selected at a
minimum!','Error Dialog Box');
elseif fieldmap == [1 1 0 0 0 0]
    % Error message for not enough parameters
    errordlg('X, Y, and Z coordinates need to be selected at a
minimum!','Error Dialog Box');
elseif fieldmap == [1 1 1 0 0 0]
    nFields = 0;
elseif fieldmap == [1 1 1 1 0 0]
    nFields = 1;
elseif fieldmap == [1 1 1 1 1 0]
    nFields = 2;
elseif fieldmap == [1 1 1 1 1 1]
    nFields = 3;
else
    % Error message for invalid parameters
    errordlg('Need to check data fields in an increasing
manner!','Error Dialog Box');
end
```

LASImport.m

```
function outfile = LASImport(infilename,nFields)
% LASImport reads in files in LAS 1.1/1.2/1.3 format.
%
% Rick Labiak
% Last Modified: 11 August 2011
% Based on LASRead Function by Cici Alexander (September 2008)
% Also based on inputs from Karl Walli (16 October 2009)
%
% INPUT
% infilename:   input file name in LAS 1.1/1.2/1.3 format
%               (for example, 'myinfile.las')
% nFields:      Default value of 0 outputs X, Y and Z coordinates of
%               the point - [X Y Z].
%               A value of 1 gives Intensity as an additional attribute
%               - [X Y Z I].
%               A value of 2 gives the Return number and the Number of
%               returns in addition to the above - [X Y Z I R N].
%               A value of 3 gives the Classification in addition to
%               the above - [X Y Z I R N C].
%
% OUTPUT
% outfile:      output matrix where each row corresponds to a single
%               return
%
% Code derived from ASPRS LAS Specifications:
% LAS Format 1.1 (7 March 2005) specifications can be found at:
```

```
% http://www.asprs.org/a/society/committees/standards/
% asprs_las_format_v11.pdf
% LAS Format 1.2 (2 September 2008) specifications can be found at:
% http://www.asprs.org/a/society/committees/standards/
% asprs_las_format_v12.pdf
% LAS Format 1.3 (14 July 2009) specifications can be found at:
% http://www.asprs.org/a/society/committees/standards/
% asprs_las_spec_v13.pdf
%
% EXAMPLE
% A = LASImport('infile.las',3)

% Open the file
fid = fopen(infilename);

% Check whether the file is valid
if fid == -1
    error('Error Opening File!')
end

% Check whether the LAS format is 1.1/1.2/1.3 - Code refers to certain
% offsets that are only correct for those versions.
fseek(fid, 24, 'bof');
VersionMajor = fread(fid,1,'uchar');
VersionMinor = fread(fid,1,'uchar');
if VersionMajor ~= 1 || (VersionMinor ~= 1 && VersionMinor ~= 2 &&
VersionMinor ~= 3)
    error('LAS Format is not 1.1, 1.2 or 1.3!')
end

% Read in the offset to point data
fseek(fid, 96, 'bof');
OffsetToPointData = fread(fid,1,'uint32');

% Read in Point Data Format ID
fseek(fid, 104, 'bof');
PointFormatID = fread(fid,1,'uchar');

% Set "offset" or "skip" values according to the specific LAS format
% and the Point Data Format ID
if VersionMinor == 1
    skip1 = 131;
    if PointFormatID == 0
        Pskip = 16;
        Iskip = 18;
        Rskip = 157;
        Cskip = 19;
    elseif PointFormatID == 1
        Pskip = 24;
        Iskip = 26;
        Rskip = 221;
        Cskip = 27;
    else
        error('Point Format ID is not compatible!');
    end
elseif VersionMinor == 2
    skip1 = 131;
```

```
    if PointFormatID == 0
        Pskip = 16;
        Iskip = 18;
        Rskip = 157;
        Cskip = 19;
    elseif PointFormatID == 1
        Pskip = 24;
        Iskip = 26;
        Rskip = 221;
        Cskip = 27;
    elseif PointFormatID == 2
        Pskip = 22;
        Iskip = 24;
        Rskip = 205;
        Cskip = 25;
    elseif PointFormatID == 3
        Pskip = 30;
        Iskip = 32;
        Rskip = 269;
        Cskip = 33;
    else
        error('Point Format ID is not compatible!');
    end
else
    skip1 = 139;
    if PointFormatID == 0
        Pskip = 16;
        Iskip = 18;
        Rskip = 157;
        Cskip = 19;
    elseif PointFormatID == 1
        Pskip = 24;
        Iskip = 26;
        Rskip = 221;
        Cskip = 27;
    elseif PointFormatID == 2
        Pskip = 22;
        Iskip = 24;
        Rskip = 205;
        Cskip = 25;
    elseif PointFormatID == 3
        Pskip = 30;
        Iskip = 32;
        Rskip = 269;
        Cskip = 33;
    else
        error('Point Format ID is not compatible!');
    end
end

% Read in the scale factors and offsets required to calculate the
% coordinates
fseek(fid, skip1, 'bof');
XScaleFactor = fread(fid,1,'double');
YScaleFactor = fread(fid,1,'double');
ZScaleFactor = fread(fid,1,'double');
XOffset = fread(fid,1,'double');
```

```
YOffset = fread(fid,1,'double');
ZOffset = fread(fid,1,'double');

% The number of bytes from the beginning of the file to the first point
% record data field is used to access the attributes of the point data
c = OffsetToPointData;

% Read in the X coordinates of the points making use of the
% XScaleFactor and XOffset values in the header
fseek(fid, c, 'bof');
X1=fread(fid,inf,'int32',Pskip);
X=X1*XScaleFactor+XOffset;

% Read in the Y coordinates of the points
fseek(fid, c+4, 'bof');
Y1=fread(fid,inf,'int32',Pskip);
Y=Y1*YScaleFactor+YOffset;

% Read in the Z coordinates of the points
fseek(fid, c+8, 'bof');
Z1=fread(fid,inf,'int32',Pskip);
Z=Z1*ZScaleFactor+ZOffset;

if nFields > 0
    % Read in the Intensity values of the points
    fseek(fid, c+12, 'bof');
    Int=fread(fid,inf,'uint16',Iskip);

    if nFields > 1
        % Read in the Return Number of the points. The first return
        % will have a return number of one, the second, two, etc.
        fseek(fid, c+14, 'bof');
        Rnum=fread(fid,inf,'bit3',Rskip);

        % Read in the Number of Returns for a given pulse.
        fseek(fid, c+14, 'bof');
        fread(fid,1,'bit3');
        Num=fread(fid,inf,'bit3',Rskip);

        if nFields > 2
            % Read in the classification values of the points
            fseek(fid, c+15, 'bof');
            Class=fread(fid,inf,'uchar',Cskip);

        end
    end
end

% Write out the file with X, Y and Z coordinates, Intensity, Return
% Number and Number of Returns depending on the fields specified in the
% input
if nFields == 0
    outfile = [X Y Z];
elseif nFields == 1
    outfile = [X Y Z Int];
elseif nFields == 2
    outfile = [X Y Z Int Rnum Num];
```

```
elseif nFields == 3
    outfile = [X Y Z Int Rnum Num Class];
end
```

Preprocessing.m

```
function varargout = Preprocessing(varargin)
% PREPROCESSING M-file for Preprocessing.fig
% Preprocessing reads in a raw point cloud, creates a DTM using a
% vendor-supplied point classification, and then references the point
% cloud to ground (creates height model or normalized DSM).
%
% Rick Labiak
% Last Modified 11 August 2011

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @Preprocessing_OpeningFcn, ...
    'gui_OutputFcn',  @Preprocessing_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before Preprocessing is made visible.
function Preprocessing_OpeningFcn(hObject, eventdata, handles,
varargin)

% Position window in screen
screen_size = get(0,'ScreenSize');
set(hObject,'Position',[screen_size(3)-1024,screen_size(4)/2-
335,1024,670]);

global my_handles
if isfield(my_handles,'roi')
    my_handles.XYZ = my_handles.roi;
else
    my_handles.XYZ = my_handles.fullcloud;
end
% Plot ROI
axes(handles.axes1);
cla
scatter3(my_handles.XYZ(:,1),my_handles.XYZ(:,2),my_handles.XYZ(:,3),2,
my_handles.XYZ(:,3),'filled')
colormap('jet')
```

```
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height ASL (m)'), title('Raw Point Cloud');
view(0,90)
rotate3d on
axis on

% Choose default command line output for Preprocessing
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = Preprocessing_OutputFcn(hObject, eventdata,
handles)
% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on button press in loadraw.
function loadraw_Callback(hObject, eventdata, handles)
global my_handles
[FileName,PathName] = uigetfile({'*.mat';'*.txt'},'Select File to
Load');
entirepath = [PathName FileName];
XYZ = importdata(entirepath);
axes(handles.axes1);
cla
% Plot ROI
scatter3(XYZ(:,1),XYZ(:,2),XYZ(:,3),2,XYZ(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height ASL (m)'), title('Raw Point Cloud');
view(0,90)
rotate3d on
axis on
my_handles.XYZ = XYZ;
guidata(hObject,handles);

% --- Executes on selection change in DTMgrid.
function DTMgrid_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function DTMgrid_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in createDTM.
function createDTM_Callback(hObject, eventdata, handles)
global my_handles
DTMRes = get(handles.DTMgrid, 'Value');
if DTMRes == 1
    my_handles.DTMRes = 1;
```

```
elseif DTMRes == 2
    my_handles.DTMRes = 2;
elseif DTMRes == 3
    my_handles.DTMRes = 5;
else
    my_handles.DTMRes = 10;
end
[my_handles.DTM,my_handles.qx,my_handles.qy,my_handles.qz] =
CreateDTM(my_handles.XYZ,my_handles.DTMRes);
% Plot DTM
axes(handles.axes1);
cla
scatter3(my_handles.DTM(:,1),my_handles.DTM(:,2),my_handles.DTM(:,3),2,
my_handles.DTM(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height ASL (m)'), title('DTM');
view(0,90)
rotate3d on
axis on
guidata(hObject,handles);

% --- Executes on button press in saveasDTM.
function saveasDTM_Callback(hObject, eventdata, handles)
global my_handles
DTM = my_handles.DTM;
qz = my_handles.qz;
qx = my_handles.qx;
qy = my_handles.qy;
[FileName,PathName] = uiputfile({'*.mat';'*.txt'},'Save As');
entirepath = [PathName FileName];
[pathstr, name, ext] = fileparts(entirepath);
if strcmp(ext,'.txt')
    dlmwrite(entirepath,DTM,'precision','%11.8g');
elseif strcmp(ext,'.mat')
    save(entirepath,'DTM','qx','qy','qz')
end

% --- Executes on button press in createHM.
function createHM_Callback(hObject, eventdata, handles)
global my_handles
[my_handles.HM] =
CreateHM(my_handles.XYZ,my_handles.qx,my_handles.qy,my_handles.qz);
% Plot HM
axes(handles.axes1);
cla
scatter3(my_handles.HM(:,1),my_handles.HM(:,2),my_handles.HM(:,3),2,my_
handles.HM(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title('Height Model');
view(0,90)
rotate3d on
```

```
axis on
guidata(hObject,handles);

function threshabove_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function threshabove_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in threshabove.
function threshabove_Callback(hObject, eventdata, handles)
global my_handles
thresha = str2double(get(handles.threshabove_edit,'String'));
indh = find(my_handles.HM(:,3) > thresha);
my_handles.HM(indh,:) = [];
% Plot
axes(handles.axes1);
cla
scatter3(my_handles.HM(:,1),my_handles.HM(:,2),my_handles.HM(:,3),2,my_
handles.HM(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title('Height Model Thresholded');
view(0,90)
rotate3d on
axis on

function threshbelow_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function threshbelow_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in threshbelow.
function threshbelow_Callback(hObject, eventdata, handles)
global my_handles
threshb = str2double(get(handles.threshbelow_edit,'String'));
indh = find(my_handles.HM(:,3) < threshb);
my_handles.HM(indh,:) = [];
% Plot
axes(handles.axes1);
cla
scatter3(my_handles.HM(:,1),my_handles.HM(:,2),my_handles.HM(:,3),2,my_
handles.HM(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title('Height Model Thresholded');
view(0,90)
rotate3d on
```

```
axis on

% --- Executes on button press in saveHM.
function saveHM_Callback(hObject, eventdata, handles)
global my_handles
HM = my_handles.HM;
[FileName,PathName] = uiputfile({'*.mat';'*.txt'},'Save As');
entirepath = [PathName FileName];
[pathstr, name, ext] = fileparts(entirepath);
if strcmp(ext,'.txt')
    dlmwrite(entirepath,HM,'precision','%11.8g');
elseif strcmp(ext,'.mat')
    save(entirepath,'HM')
end
```

CreateDTM.m

```
function [DTM,qx,qy,qz] = CreateDTM(XYZ,res)
% CreateDTM computes the Digital Terrain Model (DTM) for a raw point
% cloud so the effect of the terrain can be eventually removed.
%
% Rick Labiak
% Last Modified 11 August 2011
%
% This routine uses the vendor-supplied classification (either non-
% ground points or ground points) from column 7 to extract ground
% points.

% Delete the points from the cloud that are not classified as ground
% hits (Class 2)
Class2 = XYZ;
[ind] = find(Class2(:,7) ~= 2);
Class2(ind,:) = [];

% Since interpolation can be easily, negatively influenced by outlier
% points, a sliding window is implemented to remove outliers from local
% areas
% Define size (half size) of sliding window (must be an integer)
%win = 5;
% Define number of standard deviations to threshold above (user input)
%numstddev = 3;
% Define percentage of NaNs (user input)
%percent = 85;
% Run custom function "rmoutlierpts" - function information is found in
% function header
%[ProcClass2,outlierrow,numoutliers] =
%rmoutlierpts(Class2,win,numstddev,percent);
ProcClass2 = Class2;

% Use Matlab library function to interpolate scattered data
F = TriScatteredInterp(ProcClass2(:,1),ProcClass2(:,2),ProcClass2(:,3),
'natural');
[qx,qy] = meshgrid(min(ProcClass2(:,1)):res:max(ProcClass2(:,1)),
min(ProcClass2(:,2)):res:max(ProcClass2(:,2)));
qz = F(qx,qy);
```

```
% Arrange rasterized DTM points into a list where the 3 columns are
% X,Y,Z,respectively
[m n] = size(qz);
count = 1;
for i=1:m
    for j=1:n
       DTMx(count,1) = qx(i,j);
       DTMy(count,1) = qy(i,j);
       DTMz(count,1) = qz(i,j);
       count = count+1;
    end
end
DTM = horzcat(DTMx,DTMy,DTMz);
```

CreateHM.m

```
function [HM] = CreateHM(XYZ,qx,qy,qz)
% CreateHM computes the height model or normalized DSM by subtracting
% the DTM from a raw point cloud. Once the effect of the terrain is
% removed, point heights are referenced above ground (versus above sea
% level).
%
% Rick Labiak
% Last Modified 11 August 2011
%
% 2 options to create height model:
% Option 1 - Subset full cloud to 1st returns and interpolate - Raster
% approach
% Option 2 - Use non-Class 2 points - Point-based approach
% Note: Both methods are coded below but Option 2 is used since it
% doesn't interpolate (create false data) and the data is denser

% Option 2 -
% Delete the points from the cloud that are classified as ground hits
% (Class 2)
nonClass2 = XYZ;
[ind] = find(nonClass2(:,7) == 2);
nonClass2(ind,:) = [];

% Implement sliding window to remove outliers from local areas
% Define size (half size) of sliding window (must be an integer)
%win = 3;
% Define number of standard deviations to threshold above (user input)
%numstddev = 3;
% Define percentage of NaNs (user input)
%percent = 90;
%[ProcnonClass2,outlierrow,numoutliers] =
%rmoutlierpts(nonClass2,win,numstddev,percent);
ProcnonClass2 = nonClass2;

% Code below uses values for qx, qy, and qz that were created when
% interpolating the DTM
% Adapted code below from Dr. Murtaza Khan (searchclosest.m)
% The code below finds the location in the DTM grid (X,Y) that is
```

```
% closest/below each non-ground point, and subtracts the height of the
% ground from the height of each point
HM = ProcnonClass2(:,1:2);

for count = 1:length(ProcnonClass2)
    x = qx(1,:);
    [ignore,j] = min(abs(x-ProcnonClass2(count,1)));
    y = qy(:,1);
    [ignore,i] = min(abs(y-ProcnonClass2(count,2)));
    z = qz(i,j);
    HM(count,3) = ProcnonClass2(count,3)-z;
end

% Add remaining data fields (intensity, return number, number of
% returns,classification)
HM = horzcat(HM,ProcnonClass2(:,4:7));
```

BuildingDetect.m

```
function varargout = BuildingDetect(varargin)
% BUILDINGDETECT M-file for BuildingDetect.fig
% BuildingDetect removes high variance (vegetation) points and creates
% an initial building mask by removing points below 2 m from a
% “thinned” height model.
%
% Rick Labiak
% Last Modified 11 August 2011

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @BuildingDetect_OpeningFcn, ...
    'gui_OutputFcn',  @BuildingDetect_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before BuildingDetect is made visible.
function BuildingDetect_OpeningFcn(hObject, eventdata, handles,
varargin)

% Position window in screen
screen_size = get(0,'ScreenSize');
set(hObject,'Position',[screen_size(3)-1024,screen_size(4)/2-
335,1024,670]);
```

```
global my_handles
axes(handles.axes1);
cla
% Plot ROI
scatter3(my_handles.roi(:,1),my_handles.roi(:,2),my_handles.roi(:,3),2,
my_handles.roi(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height ASL (m)'), title('Raw Point Cloud');
view(0,90)
rotate3d on
axis on

% Choose default command line output for BuildingDetect
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = BuildingDetect_OutputFcn(hObject, eventdata,
handles)
% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on selection change in tileres.
function tileres_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function tileres_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
global my_handles
my_handles.one = 1;
my_handles.two = 1;
my_handles.three = 1;

% --- Executes on button press in calculatevar.
function calculatevar_Callback(hObject, eventdata, handles)
global my_handles
Tileres = get(handles.tileres, 'Value');
if Tileres == 1
    my_handles.tileres = 1;
    if my_handles.one == 1
        [my_handles.TileVectors1,my_handles.SortedRaw1,my_handles.m1,
        my_handles.n1] = TileCloud(my_handles.roi,my_handles.tileres);
        [my_handles.BuildingPoints1,my_handles.tileindex1] =
        TilePtVar(my_handles.TileVectors1,my_handles.SortedRaw1);
        my_handles.one = 0;
    else
    end
    % Plot
```

```
    BP1 = my_handles.BuildingPoints1;
    axes(handles.axes1);
    cla
    scatter3(BP1(:,1),BP1(:,2),BP1(:,3),2,BP1(:,3),'filled')
    colormap('jet')
    colorbar
    axis equal
    xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N
y Coordinate)'), zlabel('Height ASL (m)'), title(sprintf('%s%d','Point
Cloud w/ High Variance Removed (Tile Res = ',Tileres,' m)'));
    view(0,90)
    rotate3d on
    axis on
elseif Tileres == 2
    my_handles.tileres = 2;
    if my_handles.two == 1
        [my_handles.TileVectors2,my_handles.SortedRaw2,my_handles.m2,
        my_handles.n2] = TileCloud(my_handles.roi,my_handles.tileres);
        [my_handles.BuildingPoints2,my_handles.tileindex2] =
        TilePtVar(my_handles.TileVectors2,my_handles.SortedRaw2);
        my_handles.two = 0;
    else
    end
    % Plot
    BP2 = my_handles.BuildingPoints2;
    axes(handles.axes1);
    cla
    scatter3(BP2(:,1),BP2(:,2),BP2(:,3),2,BP2(:,3),'filled')
    colormap('jet')
    colorbar
    axis equal
    xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N
y Coordinate)'), zlabel('Height ASL (m)'), title(sprintf('%s%d','Point
Cloud w/ High Variance Removed (Tile Res = ',Tileres,' m)'));
    view(0,90)
    rotate3d on
    axis on
else
    my_handles.tileres = 3;
    if my_handles.three == 1
        [my_handles.TileVectors3,my_handles.SortedRaw3,my_handles.m3,
        my_handles.n3] = TileCloud(my_handles.roi,my_handles.tileres);
        [my_handles.BuildingPoints3,my_handles.tileindex3] =
        TilePtVar(my_handles.TileVectors3,my_handles.SortedRaw3);
        my_handles.three = 0;
    else
    end
    % Plot
    BP3 = my_handles.BuildingPoints3;
    axes(handles.axes1);
    cla
    scatter3(BP3(:,1),BP3(:,2),BP3(:,3),2,BP3(:,3),'filled')
    colormap('jet')
    colorbar
    axis equal
    xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N
y Coordinate)'), zlabel('Height ASL (m)'), title(sprintf('%s%d','Point
```

```
Cloud w/ High Variance Removed (Tile Res = ',Tileres,' m)'));
    view(0,90)
    rotate3d on
    axis on
end
guidata(hObject, handles);

% --- Executes on selection change in gausssize.
function gausssize_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function gausssize_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in calculategauss.
function calculategauss_Callback(hObject, eventdata, handles)
global my_handles
GaussSize = get(handles.gausssize, 'Value');
TileRes = my_handles.tileres;
if TileRes == 1
    my_handles.TileVectors = my_handles.TileVectors1;
    my_handles.tileindex = my_handles.tileindex1;
    my_handles.SortedRaw = my_handles.SortedRaw1;
    my_handles.m = my_handles.m1;
    my_handles.n = my_handles.n1;
elseif TileRes == 2
    my_handles.TileVectors = my_handles.TileVectors2;
    my_handles.tileindex = my_handles.tileindex2;
    my_handles.SortedRaw = my_handles.SortedRaw2;
    my_handles.m = my_handles.m2;
    my_handles.n = my_handles.n2;
else
    my_handles.TileVectors = my_handles.TileVectors3;
    my_handles.tileindex = my_handles.tileindex3;
    my_handles.SortedRaw = my_handles.SortedRaw3;
    my_handles.m = my_handles.m3;
    my_handles.n = my_handles.n3;
end
if GaussSize == 1
    my_handles.gaussfiltsize = 2;
    [my_handles.BuildingPointsGauss] =
GaussFiltRemTiles(my_handles.gaussfiltsize,my_handles.TileVectors,my_ha
ndles.SortedRaw,my_handles.tileindex,my_handles.m,my_handles.n);
elseif GaussSize == 2
    my_handles.gaussfiltsize = 3;
    [my_handles.BuildingPointsGauss] =
GaussFiltRemTiles(my_handles.gaussfiltsize,my_handles.TileVectors,my_ha
ndles.SortedRaw,my_handles.tileindex,my_handles.m,my_handles.n);
elseif GaussSize == 3
    my_handles.gaussfiltsize = 4;
    [my_handles.BuildingPointsGauss] =
GaussFiltRemTiles(my_handles.gaussfiltsize,my_handles.TileVectors,my_ha
ndles.SortedRaw,my_handles.tileindex,my_handles.m,my_handles.n);
else
    my_handles.gaussfiltsize = 5;
```

```
    [my_handles.BuildingPointsGauss] =
GaussFiltRemTiles(my_handles.gaussfiltsize,my_handles.TileVectors,my_ha
ndles.SortedRaw,my_handles.tileindex,my_handles.m,my_handles.n);
end
% Plot
BPG = my_handles.BuildingPointsGauss;
axes(handles.axes1);
cla
scatter3(BPG(:,1),BPG(:,2),BPG(:,3),2,BPG(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height ASL (m)'),
title(sprintf('%s%d%s%d','Variance Removed (Tile Res = ',TileRes,' m
and Gaussian Win = ',GaussSize+1,' )'));
view(0,90)
rotate3d on
axis on

function thresh_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function thresh_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in thresholdheight.
function thresholdheight_Callback(hObject, eventdata, handles)
global my_handles
threshouth = str2double(get(handles.thresh_edit,'String'));
my_handles.BuildingPtGaussThresh = my_handles.BuildingPointsGauss;
indh = find(my_handles.BuildingPtGaussThresh(:,3) > threshouth);
my_handles.BuildingPtGaussThresh(indh,:) = [];
% Plot
BPGT = my_handles.BuildingPtGaussThresh;
axes(handles.axes1);
cla
scatter3(BPGT(:,1),BPGT(:,2),BPGT(:,3),2,BPGT(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height ASL (m)'), title(sprintf('%s%d','High
Variance Removed and Threshold Above ',threshouth,' m'));
view(0,90)
rotate3d on
axis on

% --- Executes on button press in createhm_bldg.
function createhm_bldg_Callback(hObject, eventdata, handles)
global my_handles
[my_handles.HMBldg] =
CreateHM(my_handles.BuildingPtGaussThresh,my_handles.qx,my_handles.qy,m
y_handles.qz);
% Plot HM
```

```
axes(handles.axes1);
cla
scatter3(my_handles.HMBldg(:,1),my_handles.HMBldg(:,2),my_handles.HMBld
g(:,3),2,my_handles.HMBldg(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title('Height Model');
view(0,90)
rotate3d on
axis on
guidata(hObject,handles);

function threshb_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function threshb_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in threshheightb.
function threshheightb_Callback(hObject, eventdata, handles)
global my_handles
threshbelow = str2double(get(handles.threshb_edit,'String'));
my_handles.HMBldgThresh = my_handles.HMBldg;
indh = find(my_handles.HMBldgThresh(:,3) < threshbelow);
my_handles.HMBldgThresh(indh,:) = [];
% Plot
axes(handles.axes1);
cla
scatter3(my_handles.HMBldgThresh(:,1),my_handles.HMBldgThresh(:,2),my_h
andles.HMBldgThresh(:,3),2,my_handles.HMBldgThresh(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title(sprintf('%s%d','Height Model
w/ Points Below ',threshbelow,' m Removed'));
view(0,90)
rotate3d on
axis on

function ZMres_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function ZMres_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in createZMs.
function createZMs_Callback(hObject, eventdata, handles)
global my_handles
my_handles.ZMres = str2double(get(handles.ZMres_edit,'String'));
[my_handles.ZM_maxheight,my_handles.ZM_normintens,my_handles.ZM_normh,m
```

```
y_handles.ZM_multret] =
CreateZMs(my_handles.HMBldgThresh,my_handles.ZMres);
eval('BuildingSegmentation')
```

TileCloud.m

```
function [TileVectors,SortedRaw,m,n,tilecount] = TileCloud(Raw,res)
% TileCloud tiles a point cloud based on a grid size determined by the
% user.
%
% Rick Labiak
% Last Modified 11 August 2011

maxX = max(Raw(:,1));
minX = min(Raw(:,1));
maxY = max(Raw(:,2));
minY = min(Raw(:,2));
xrange = maxX-minX;
yrange = maxY-minY;

% Subtract remainder from extent so entire area can be evenly divided
% into grids
Rx = mod(xrange,res);
Ry = mod(yrange,res);
xrange = xrange - Rx;
yrange = yrange - Ry;
i = 0:res:yrange;
j = 0:res:xrange;
ytemp = i+minY;
xtemp = j+minX;

% Sort points by X, then Y
SortedRaw = sortrows(Raw,[1 2]);
% Add "index" column to SortedRaw points
index = 1:size(SortedRaw,1);
SortedRaw = horzcat(SortedRaw,index');
clear index;

% Create cell array and initialize to zero
TileVectors = cell((length(i)-1)*(length(j)-1),1);

% Initialize counter
count = 1;

tilecount = zeros(1,length(TileVectors));

for m=1:length(i)-1
    for n=1:length(j)-1
        index = SortedRaw(:,1) > xtemp(n) & SortedRaw(:,1) < xtemp(n+1)
& SortedRaw(:,2) > ytemp(m) & SortedRaw(:,2) < ytemp(m+1);
        TileVectors{count,1} = SortedRaw(index,:);
        tilecount(count) = sum(index);
        clear index
        count = count + 1;
    end
```

```
end

m = length(ytemp)-1;
n = length(xtemp)-1;
```

TilePtVar.m

```
function [BuildingPoints,tileindex] = TilePtVar(TileVectors,SortedRaw)
% TilePtVar calculates the height variance in each tile and determines
% the outlier tiles in order to remove tree points (vegetation).
%
% Rick Labiak
% Last Modified 11 August 2011
Var = zeros(length(TileVectors),1);
Remove = [];

for i = 1:length(TileVectors)
    XYZ = TileVectors{i,1};
    if isempty(XYZ)
    else
        Var(i) = std(XYZ(:,3))^2;
    end
end

% First remove outliers
varavg = mean(Var);
varstd = std(Var);
upperthreshout = varavg + 3*varstd;

TempVar = Var;
TempVar(Var > upperthreshout) = [];
varavg = mean(TempVar);
varstd = std(TempVar);
upperthresh = varavg + 2*varstd;

tileindex = find(Var > upperthresh);

for i = 1:length(tileindex)
    Temp = TileVectors{tileindex(i),1};
    RemoveTile = Temp(:,8);
    Remove = vertcat(Remove,RemoveTile);
end

BuildingPoints = SortedRaw;
BuildingPoints(Remove,:) = [];
```

GaussFiltRemTiles.m

```
function [BuildingPoints] =
GaussFiltRemTiles(gaussfiltsize,TileVectors,SortedRaw,tileindex,m,n)
% GaussFiltRemTiles removes tiles surrounded by already removed tiles.
%
% Rick Labiak
```

```
% Last Modified 11 August 2011

Remove2 = [];

% Creates removed tiles mask
TileMatrix = ones(m,n)';
TileMatrix(tileindex) = 0;
TileMatrix = TileMatrix';

% Gaussian blurs mask
h = fspecial('gaussian',gaussfiltsize);
g = imfilter(TileMatrix,h,'replicate');

TileMatrix(g < 1) = 0;

TileMatrix = TileMatrix';
newtileindex = reshape(TileMatrix,m*n,1);

ind = find(newtileindex == 0);

for i = 1:length(ind)
    Temp = TileVectors{ind(i),1};
    Remove2Tile = Temp(:,8);
    Remove2 = vertcat(Remove2,Remove2Tile);
end

BuildingPoints = SortedRaw;
BuildingPoints(Remove2,:) = [];
```

CreateZMs.m

```
function [ZM_maxheight,ZM_normintens,ZM_normh,ZM_multret] =
CreateZMs(HM,res)
% CreateZMs computes the metric images for a point cloud based on a
% user-defined grid resolution.
%
% Rick Labiak
% Last Modified 11 August 2011

maxX = max(HM(:,1));
minX = min(HM(:,1));
maxY = max(HM(:,2));
minY = min(HM(:,2));
xrange = maxX-minX;
yrange = maxY-minY;

% Subtract remainder from extent so entire area can be evenly divided %
into grids
Rx = mod(xrange,res);
Ry = mod(yrange,res);
xrange = xrange - Rx;
yrange = yrange - Ry;

% Sort points by X, then Y
SortedXYZ = sortrows(HM,[1 2]);
```

```
% Initialize new zonal attribute matrices
ZM_maxheight = zeros(yrange/res,xrange/res);
ZM_normintens = zeros(yrange/res,xrange/res);
ZM_normh = zeros(yrange/res,xrange/res);

i = 0:res:yrange;
j = 0:res:xrange;
ytemp = i+minY;
xtemp = j+minX;

for m=1:length(i)-1
    for n=1:length(j)-1
        index = find(SortedXYZ(:,1) > xtemp(n) & SortedXYZ(:,1) <
xtemp(n+1) & SortedXYZ(:,2) > ytemp(m) & SortedXYZ(:,2) < ytemp(m+1));
        if isempty(index)
            ZM_maxheight(m,n) = 0;
            ZM_normintens(m,n) =
mean(SortedXYZ(index,4)/max(SortedXYZ(index,4)));
            ZM_normh(m,n) =
mean(SortedXYZ(index,3)/max(SortedXYZ(index,3)));
        else
            ZM_maxheight(m,n) = max(SortedXYZ(index,3));
            ZM_normintens(m,n) =
mean(SortedXYZ(index,4)/max(SortedXYZ(index,4)));
            ZM_normh(m,n) =
mean(SortedXYZ(index,3)/max(SortedXYZ(index,3)));
        end
        clear index
    end
end

% Find total number of multiple returns in each zone
% Find/delete points that ARE NOT multiple returns...leaving only those
% multiple return points
SortedXYZ = sortrows(HM,[1 2]);
[index] = find(SortedXYZ(:,6) ~= 2);
SortedXYZ(index,:) = [];
clear index

% Initialize new zonal attribute matrix
ZM_multret = zeros(yrange/res,xrange/res);

i = 0:res:yrange;
j = 0:res:xrange;
ytemp = i+minY;
xtemp = j+minX;

for m=1:length(i)-1
    for n=1:length(j)-1
        index = find(SortedXYZ(:,1) > xtemp(n) & SortedXYZ(:,1) <
xtemp(n+1) & SortedXYZ(:,2) > ytemp(m) & SortedXYZ(:,2) < ytemp(m+1));
        ZM_multret(m,n) = length(index);
        clear index
    end
end
```

BuildingSegmentation.m

```
function varargout = BuildingSegmentation(varargin)
% BUILDINGSEGMENTATION M-file for BuildingSegmentation.fig
% BuildingSegmentation further defines building segments by
% thresholding and filtering metric images and masking out the multiple
% returns.
%
% Rick Labiak
% Last Modified 11 August 2011

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @BuildingSegmentation_OpeningFcn, ...
    'gui_OutputFcn',  @BuildingSegmentation_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before BuildingSegmentation is made visible.
function BuildingSegmentation_OpeningFcn(hObject, eventdata, handles,
varargin)

% Position window in screen
screen_size = get(0,'ScreenSize');
set(hObject,'Position',[screen_size(3)-1024,screen_size(4)/2-
384,1024,768]);

global my_handles
axes(handles.axes2);
cla
scatter3(my_handles.HMBldgThresh(:,1),my_handles.HMBldgThresh(:,2),my_h
andles.HMBldgThresh(:,3),2,my_handles.HMBldgThresh(:,3),'filled')
colormap('jet')
colorbar
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title('Height Model - Point
Cloud');
view(0,90)
rotate3d on
axis on

ZMType = get(handles.plotZM, 'Value');
if ZMType == 1
```

```
    my_handles.currentZM = my_handles.ZM_normh;
elseif ZMType == 2
    my_handles.currentZM = my_handles.ZM_normintens;
else
    my_handles.currentZM = my_handles.ZM_maxheight;
end

my_handles.xax = my_handles.HMBldgThresh(:,1);
my_handles.yax = my_handles.HMBldgThresh(:,2);
[my_handles.l my_handles.w] = size(my_handles.ZM_normh);

% Plot selected ZM
axes(handles.axes1);
cla
imagesc(my_handles.currentZM)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Metric
Image');
axis xy
axis equal
colorbar

% Choose default command line output for BuildingSegmentation
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = BuildingSegmentation_OutputFcn(hObject, eventdata,
handles)
varargout{1} = handles.output;

% --- Executes on selection change in plotZM.
function plotZM_Callback(hObject, eventdata, handles)
global my_handles
ZMType = get(handles.plotZM, 'Value');
if ZMType == 1
    my_handles.currentZM = my_handles.ZM_normh;
elseif ZMType == 2
    my_handles.currentZM = my_handles.ZM_normintens;
else
    my_handles.currentZM = my_handles.ZM_maxheight;
end
% Plot selected ZM
axes(handles.axes1);
cla
imagesc(my_handles.currentZM)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Metric
Image');
axis xy
axis equal
colorbar

% --- Executes during object creation, after setting all properties.
function plotZM_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
```

```
    set(hObject,'BackgroundColor','white');
end

function threshZM_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function threshZM_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes during object creation, after setting all properties.
function threshZM_CreateFcn(hObject, eventdata, handles)

% --- Executes on button press in threshZM.
function threshZM_Callback(hObject, eventdata, handles)
global my_handles
threshZM = str2double(get(handles.threshZM_edit,'String'));
my_handles.currentZM(my_handles.currentZM < threshZM) = 0;
% Plot thresholded ZM
axes(handles.axes1);
cla
imagesc(my_handles.currentZM)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Metric
Image');
axis xy
axis equal
colorbar

function filtersize_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes during object creation, after setting all properties.
function filterZM_CreateFcn(hObject, eventdata, handles)
% --- Executes on button press in filterZM.
function filterZM_Callback(hObject, eventdata, handles)
global my_handles
index = get(handles.filtersize, 'Value');
if index == 1
    FilterKernelSize = 2;
elseif index == 2
    FilterKernelSize = 3;
elseif index == 3
    FilterKernelSize = 4;
else
    FilterKernelSize = 5;
end
h = fspecial('gaussian',FilterKernelSize);
my_handles.currentZM = imfilter(my_handles.currentZM,h,'replicate');
% Plot thresholded ZM
axes(handles.axes1);
cla
imagesc(my_handles.currentZM)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Metric
```

```
Image');
axis xy
axis equal
colorbar

% --- Executes during object creation, after setting all properties.
function horiz_space1_check_CreateFcn(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function fillZM_CreateFcn(hObject, eventdata, handles)
% --- Executes on button press in fillZM.
function fillZM_Callback(hObject, eventdata, handles)
global my_handles
if (get(handles.horiz_space1_check,'Value') ==
get(handles.horiz_space1_check,'Max'))
    my_handles.space = 1;
    my_handles.flag = 0;
end
if (get(handles.horiz_space2_check,'Value') ==
get(handles.horiz_space2_check,'Max'))
    my_handles.space = 2;
    my_handles.flag = 0;
end
if (get(handles.horiz_space3_check,'Value') ==
get(handles.horiz_space3_check,'Max'))
    my_handles.space = 3;
    my_handles.flag = 0;
end
if (get(handles.vert_space1_check,'Value') ==
get(handles.vert_space1_check,'Max'))
    my_handles.space = 1;
    my_handles.flag = 1;
end
if (get(handles.vert_space2_check,'Value') ==
get(handles.vert_space2_check,'Max'))
    my_handles.space = 2;
    my_handles.flag = 1;
end
if (get(handles.vert_space3_check,'Value') ==
get(handles.vert_space3_check,'Max'))
    my_handles.space = 3;
    my_handles.flag = 1;
end
my_handles.currentZM =
PixelFill(my_handles.currentZM,my_handles.space,my_handles.flag);
% Plot filled ZM
axes(handles.axes1);
cla
imagesc(my_handles.currentZM)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Metric
Image');
axis xy
axis equal
colorbar

% --- Executes during object creation, after setting all properties.
function SaveZMpopup_CreateFcn(hObject, eventdata, handles)
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
% --- Executes during object creation, after setting all properties.
function saveZM_CreateFcn(hObject, eventdata, handles)
% --- Executes on button press in saveZM.
function saveZM_Callback(hObject, eventdata, handles)
global my_handles
ind = get(handles.SaveZMpopup, 'Value');
if ind == 1
    my_handles.A = my_handles.currentZM;
elseif ind == 2
    my_handles.B = my_handles.currentZM;
else
    my_handles.C = my_handles.currentZM;
end

% --- Executes on button press in multret_check.
function multret_check_Callback(hObject, eventdata, handles)
global my_handles
my_handles.currentMR = my_handles.ZM_multret;
% Plot multiple return ZM in main axes
axes(handles.axes1);
cla
imagesc(my_handles.currentMR)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Multiple
Returns Metric Image');
axis xy
axis equal
colorbar

function threshmr_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function threshmr_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in threshmr.
function threshmr_Callback(hObject, eventdata, handles)
global my_handles
threshMR = str2double(get(handles.threshmr_edit,'String'));
my_handles.currentMR(my_handles.currentMR <= threshMR) = 0;
% Plot thresholded ZM
axes(handles.axes1);
cla
imagesc(my_handles.currentMR)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Multiple
Returns Image - Thresholded');
axis xy
axis equal
colorbar

% --- Executes on button press in filtmr.
function filtmr_Callback(hObject, eventdata, handles)
```

```
global my_handles
index = get(handles.filtmrsize, 'Value');
if index == 1
    FilterKernelSize = 2;
elseif index == 2
    FilterKernelSize = 3;
elseif index == 3
    FilterKernelSize = 4;
else
    FilterKernelSize = 5;
end
h = fspecial('gaussian',FilterKernelSize);
my_handles.currentMR = imfilter(my_handles.currentMR,h,'replicate');
% Plot
axes(handles.axes1);
cla
imagesc(my_handles.currentMR)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Multiple
Returns Image - Filtered');
axis xy
axis equal
colorbar

% --- Executes on selection change in filtmrsize.
function filtmrsize_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function filtmrsize_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on selection change in subtractmenu.
function subtractmenu_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
global my_handles
multret_mask = zeros(size(my_handles.currentMR));
multret_mask(my_handles.currentMR > 0) = 1;
index = get(handles.subtractmenu, 'Value');
if index == 1
    my_handles.currentsub = my_handles.ZM_normh;
    my_handles.currentsub(multret_mask == 1) = 0;
elseif index == 2
    my_handles.currentsub = my_handles.ZM_normintens;
    my_handles.currentsub(multret_mask == 1) = 0;
elseif index == 3
    my_handles.currentsub = my_handles.ZM_maxheight;
    my_handles.currentsub(multret_mask == 1) = 0;
elseif index == 4
    my_handles.currentsub = my_handles.A;
    my_handles.currentsub(multret_mask == 1) = 0;
elseif index == 5
    my_handles.currentsub = my_handles.B;
    my_handles.currentsub(multret_mask == 1) = 0;
else
    my_handles.currentsub = my_handles.C;
    my_handles.currentsub(multret_mask == 1) = 0;
```

```
end
% Plot Result
axes(handles.axes1);
cla
imagesc(my_handles.currentsub)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Multiple
Returns Masked');
axis xy
axis equal
colorbar

function subtractmenu_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in savesub.
function savesub_Callback(hObject, eventdata, handles)
global my_handles
ind = get(handles.savemrmenu, 'Value');
if ind == 1
    my_handles.D = my_handles.currentsub;
elseif ind == 2
    my_handles.E = my_handles.currentsub;
else
    my_handles.F = my_handles.currentsub;
end

% --- Executes on selection change in savemrmenu.
function savemrmenu_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function savemrmenu_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on selection change in plotselect.
function plotselect_Callback(hObject, eventdata, handles)
global my_handles
ind = get(handles.plotselect, 'Value');
if ind == 2
    my_handles.currentplot = my_handles.ZM_normh;
elseif ind == 3
    my_handles.currentplot = my_handles.ZM_normintens;
elseif ind == 4
    my_handles.currentplot = my_handles.ZM_maxheight;
elseif ind == 5
    my_handles.currentplot = my_handles.A;
elseif ind == 6
    my_handles.currentplot = my_handles.B;
elseif ind == 7
    my_handles.currentplot = my_handles.C;
elseif ind == 8
    my_handles.currentplot = my_handles.D;
elseif ind == 9
```

```
    my_handles.currentplot = my_handles.E;
elseif ind == 10
    my_handles.currentplot = my_handles.F;
else
end
% Plot Selection
axes(handles.axes1);
cla
imagesc(my_handles.currentplot)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Selected
Plot');
axis xy
axis equal
colorbar

% --- Executes during object creation, after setting all properties.
function plotselect_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in nextpagebutton.
function nextpagebutton_Callback(hObject, eventdata, handles)
eval('BuildingMorphology')
```

PixelFill.m

```
function [I] = PixelFill(I,space,flag)
% PixelFill fills in empty pixels with the average value of surrounding
% pixels. Inputs include the direction to be filled (either horizontal
% or vertical) and the number of consecutive empty pixels to be filled.
%
% Rick Labiak
% Last Modified 11 August 2011

I(isnan(I)) = 0;
[m n] = size(I);

if flag == 0;
    for i=1:m
        for j=1:n-space-1
            local = I(i,j:j+space+1);
            if local(2:space+1) == 0
                if (local(1) ~= 0 && local(end) ~= 0)
                    local(2:space+1) = mean([local(1),local(end)]);
                    I(i,j:j+space+1) = local;
                else
                end
            else
            end
        end
    end
else
    I = I';
```

```
    [m n] = size(I);
    for i=1:m
        for j=1:n-space-1
            local = I(i,j:j+space+1);
            if local(2:space+1) == 0
                if (local(1) ~= 0 && local(end) ~= 0)
                    local(2:space+1) = mean([local(1),local(end)]);
                    I(i,j:j+space+1) = local;
                else
                end
            else
            end
        end
    end
    I = I';
end
```

BuildingMorphology.m

```
function varargout = BuildingMorphology(varargin)
% BUILBoundsINGMORPHOLOGY M-file for BuildingMorphology.fig
% BuildingMorphology finalizes and extracts individual building regions
% using morphological processing and connected components analysis.
%
% Rick Labiak
% Last Modified 11 August 2011

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @BuildingMorphology_OpeningFcn, ...
    'gui_OutputFcn',  @BuildingMorphology_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before BuildingMorphology is made visible.
function BuildingMorphology_OpeningFcn(hObject, eventdata, handles,
varargin)
global my_handles

% Position window in screen
screen_size = get(0,'ScreenSize');
set(hObject,'Position',[screen_size(3)-1024,screen_size(4)/2-
384,1024,768]);
```

```
% Plot Picture of Site
[pathstr, my_handles.name, ext, versn] =
fileparts(my_handles.fullpath);
my_handles.siteImage = imread([my_handles.name,'.png']);
axes(handles.axes2);
cla reset
set(handles.axes2,'Position',[1012-my_handles.w,264-
.5*my_handles.l,my_handles.w,my_handles.l]);
imagesc(my_handles.xax,my_handles.yax,my_handles.siteImage)
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), title([my_handles.name,' Site']);
axis off

% Choose default command line output for BuildingMorphology
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = BuildingMorphology_OutputFcn(hObject, eventdata,
handles)

% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on selection change in selectimage.
function selectimage_Callback(hObject, eventdata, handles)
global my_handles
ind = get(handles.selectimage, 'Value');
if ind == 2
    my_handles.currentim = my_handles.ZM_normh;
elseif ind == 3
    my_handles.currentim = my_handles.ZM_normintens;
elseif ind == 4
    my_handles.currentim = my_handles.ZM_maxheight;
elseif ind == 5
    my_handles.currentim = my_handles.A;
elseif ind == 6
    my_handles.currentim = my_handles.B;
elseif ind == 7
    my_handles.currentim = my_handles.C;
elseif ind == 8
    my_handles.currentim = my_handles.D;
elseif ind == 9
    my_handles.currentim = my_handles.E;
elseif ind == 10
    my_handles.currentim = my_handles.F;
else
end
% Plot Selection
axes(handles.axes1);
cla
imagesc(my_handles.currentim)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Image to
Segment');
```

```
axis equal
axis xy
colorbar

% --- Executes during object creation, after setting all properties.
function selectimage_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in erosion.
function erosion_Callback(hObject, eventdata, handles)

% --- Executes on button press in opening.
function opening_Callback(hObject, eventdata, handles)

% --- Executes on button press in closing.
function closing_Callback(hObject, eventdata, handles)

function radius_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function radius_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function height_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function height_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function width_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function width_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in morph_process.
function morph_process_Callback(hObject, eventdata, handles)
global my_handles
if (get(handles.erosion,'Value') == get(handles.erosion,'Max'))
    my_handles.operation = 1;
end
if (get(handles.opening,'Value') == get(handles.opening,'Max'))
    my_handles.operation = 2;
end
if (get(handles.closing,'Value') == get(handles.closing,'Max'))
    my_handles.operation = 3;
end
if (get(handles.ball,'Value') == get(handles.ball,'Max'))
```

```
    my_handles.type = 1;
end
if (get(handles.square,'Value') == get(handles.square,'Max'))
    my_handles.type = 2;
end
my_handles.morphrad = str2double(get(handles.radius_edit,'String'));
my_handles.morphh = str2double(get(handles.height_edit,'String'));
my_handles.morphw = str2double(get(handles.width_edit,'String'));
[my_handles.currentim] =
MorphProcess(my_handles.currentim,my_handles.operation,my_handles.type,
my_handles.morphw,my_handles.morphrad,my_handles.morphh);
% Plot Result
axes(handles.axes1);
cla
imagesc(my_handles.currentim)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Image After
Morphological Processing');
axis equal
axis xy
colorbar

function morphthresh_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function morphthresh_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in morphthresh.
function morphthresh_Callback(hObject, eventdata, handles)
global my_handles
thresh = str2double(get(handles.morphthresh_edit,'String'));
my_handles.currentim(my_handles.currentim < thresh) = 0;
% Plot Result
axes(handles.axes1);
cla
imagesc(my_handles.currentim)
xlabel('x Location (m)'), ylabel('y Location (m)'), title('Image After
Morphological Processing/Thresholding');
axis equal
axis xy
colorbar

% --- Executes on button press in extract.
function extract_Callback(hObject, eventdata, handles)
global my_handles
% Extract ROIs
bldg_mask = my_handles.currentim;
bldg_mask(my_handles.currentim ~= 0) = 1;
my_handles.uniquebldgs = bwlabel(bldg_mask,4);
my_handles.num = max(max(my_handles.uniquebldgs));
my_handles.uniquebldgs(my_handles.uniquebldgs == 0) = my_handles.num+1;
% Plot Result
axes(handles.axes1);
cla
imagesc(my_handles.uniquebldgs)
```

```
xlabel('x Location (m)'), ylabel('y Location (m)'),
title(sprintf('%s%d','Building Regions (',my_handles.num,' Total)'));
axis equal
axis xy
myColorMap = jet(my_handles.num+1); % Make a copy of jet.
% Assign white (all 1's) to black (the first row in myColorMap).
myColorMap(my_handles.num+1, :) = [1 1 1];
colormap(myColorMap); % Apply the colormap
colorbar

% Get boundaries of regions
XYZH = my_handles.HM;
minX = min(XYZH(:,1));
minY = min(XYZH(:,2));
[B,L] = bwboundaries(bldg_mask,4,'noholes');
% Change boundary indicies to lat/long coordinates
my_handles.Bounds = B;
for i = 1:length(B)
    my_handles.Bounds{i}(:,1) = B{i}(:,1)+minY;
    my_handles.Bounds{i}(:,2) = B{i}(:,2)+minX;
end
```

MorphProcess.m

```
function [g] = MorphProcess(I,operation,type,w,r,h)
% MorphProcess performs morphological operations on a binary image
% based on user inputs for operation type, structuring element, and
% size.
%
% Rick Labiak
% Last Modified 11 August 2011

I(isnan(I)) = 0;

if type == 1
    se = strel('ball', r, h);
else
    se = strel('square',w);
end

if operation == 1
    g = imerode(I,se);
elseif operation == 2
    g = imopen(I,se);
else
    g = imclose(I,se);
end

g = mat2gray(g);
```

DamageDetectionFINAL.m

```
function varargout = DamageDetectionFINAL(varargin)
% DAMAGEDETECTIONFINAL M-file for DamageDetectionFINAL.fig
```

```
% DamageDetectionFINAL performs damage assessment using normal vector
% and height variance metrics to produce initial damage assessment and
% damage percent maps.
%
% Rick Labiak
% Last Modified 11 August 2011

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @DamageDetectionFINAL_OpeningFcn, ...
    'gui_OutputFcn',  @DamageDetectionFINAL_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before DamageDetectionFINAL is made visible.
function DamageDetectionFINAL_OpeningFcn(hObject, eventdata, handles,
varargin)

% Position window in screen
screen_size = get(0,'ScreenSize');
set(hObject,'Position',[screen_size(3)-1024,screen_size(4)/2-
384,1024,768]);

% Choose default command line output for DamageDetectionFINAL
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = DamageDetectionFINAL_OutputFcn(hObject, eventdata,
handles)

% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on button press in bldgregselect.
function bldgregselect_Callback(hObject, eventdata, handles)
global my_handles
% Select Pixel
if my_handles.shapeflag == 0
    axes(handles.axes2);
    P = impixel();
    my_handles.i = P(1);
    IN =
```

```
inpolygon(my_handles.HM(:,1),my_handles.HM(:,2),my_handles.Bounds{my_ha
ndles.i}(:,2),my_handles.Bounds{my_handles.i}(:,1));
    my_handles.Region = my_handles.HM(IN,:);
else
    axes(handles.axes2);
    P = impixel();
    my_handles.i = P(1);
    ind = my_handles.shapeallregs(:,8) == my_handles.i;
    my_handles.Region = my_handles.shapeallregs(ind,:);
end
% Plot building region
axes(handles.axes1);
cla
plot3(my_handles.Region(:,1),my_handles.Region(:,2),my_handles.Region(:
,3),'.k')
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title(sprintf('%s%d','Building
Region ',my_handles.i,' Points'));
view(-30,20)
rotate3d on

% --- Executes on button press in resetpage.
function reset_Callback(hObject, eventdata, handles)
axes(handles.axes1);
cla reset
set(handles.normdampts,'String',' ');
set(handles.tileptdensity,'String',' ');
set(handles.vardampts,'String',' ');
global my_handles
% Select Pixel
if my_handles.shapeflag == 0
    axes(handles.axes2);
    P = impixel();
    my_handles.i = P(1);
    IN =
inpolygon(my_handles.HM(:,1),my_handles.HM(:,2),my_handles.Bounds{my_ha
ndles.i}(:,2),my_handles.Bounds{my_handles.i}(:,1));
    my_handles.Region = my_handles.HM(IN,:);
else
    axes(handles.axes2);
    P = impixel();
    my_handles.i = P(1);
    ind = my_handles.shapeallregs(:,8) == my_handles.i;
    my_handles.Region = my_handles.shapeallregs(ind,:);
end
% Plot building region
axes(handles.axes1);
cla
plot3(my_handles.Region(:,1),my_handles.Region(:,2),my_handles.Region(:
,3),'.k')
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title(sprintf('%s%d','Building
Region ',my_handles.i,' Points'));
view(-30,20)
rotate3d on
```

```
function tilesize_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function tilesize_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in tilenormals.
function tilenormals_Callback(hObject, eventdata, handles)
global my_handles
my_handles.tilesize = str2double(get(handles.tilesize_edit,'String'));
[my_handles.RegionTiles,SortedRegion,m,n,tilecount] =
TileCloud(my_handles.Region,my_handles.tilesize);
tilecount(tilecount == 0) = nan;
AvgTileCount = round(nanmean(tilecount));
set(handles.tileptdensity,'String',AvgTileCount);

[my_handles.IndDamPts,OffNormals,OffXYZ,OnNormals,OnXYZ,my_handles.Metr
ic1,my_handles.Metric2] =
TileNormDamage(my_handles.RegionTiles,my_handles.Region);

% Plot points and normal vectors
axes(handles.axes1);
cla
plot3(my_handles.Region(:,1),my_handles.Region(:,2),my_handles.Region(:
,3),'.k')
if isempty(OnNormals)
else
    hold on

quiver3(OnXYZ(1,:)',OnXYZ(2,:)',OnXYZ(3,:)',OnNormals(1,:)',OnNormals(2
,:)',OnNormals(3,:)','-b')
end
if isempty(OffNormals)
else
    hold on

quiver3(OffXYZ(1,:)',OffXYZ(2,:)',OffXYZ(3,:)',OffNormals(1,:)',OffNorm
als(2,:)',OffNormals(3,:)','-r')
end
if isempty(my_handles.IndDamPts)
else
    hold on

plot3(my_handles.IndDamPts(:,1),my_handles.IndDamPts(:,2),my_handles.In
dDamPts(:,3),'.r')
end
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title(sprintf('%s%d','Building
Region ',my_handles.i,' Points'));
view(-30,20)
rotate3d on

Metric1 = round(my_handles.Metric1*100);
```

```
Metric2 = round(my_handles.Metric2*100);
set(handles.normdampts,'String',Metric2);

function tilesizevar_edit_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function tilesizevar_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in tilevariance.
function tilevariance_Callback(hObject, eventdata, handles)
global my_handles
my_handles.tilesizevar =
str2double(get(handles.tilesizevar_edit,'String'));
[my_handles.RegionTiles,SortedRegion,m,n,tilecount] =
TileCloud(my_handles.Region,my_handles.tilesizevar);
[my_handles.DamPtsTileVar,my_handles.Metric3] =
DamDetVar(my_handles.RegionTiles,my_handles.Region);
axes(handles.axes1);
cla
plot3(my_handles.Region(:,1),my_handles.Region(:,2),my_handles.Region(:
,3),'.k')
if isempty(my_handles.DamPtsTileVar)
else
    hold on
plot3(my_handles.DamPtsTileVar(:,1),my_handles.DamPtsTileVar(:,2),my_ha
ndles.DamPtsTileVar(:,3),'.r')
end
axis equal
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), zlabel('Height (m)'), title(sprintf('%s%d','Building
Region ',my_handles.i,' Points'));
view(-30,20)
rotate3d on

Metric3 = round(my_handles.Metric3*100);
set(handles.vardampts,'String',Metric3);

% --- Executes on button press in tilenormcheck.
function tilenormcheck_Callback(hObject, eventdata, handles)

% --- Executes on button press in tilevarcheck.
function tilevarcheck_Callback(hObject, eventdata, handles)

% --- Executes on button press in origbldgreg.
function origbldgreg_Callback(hObject, eventdata, handles)
global my_handles
my_handles.shapeflag = 0;
axes(handles.axes2);
cla reset
imagesc(my_handles.uniquebldgs)
xlabel('x Location (m)'), ylabel('y Location (m)'),
title(sprintf('%s%d','Building Regions (',my_handles.num,' Total)'));
axis xy
axis equal
```

```
myColorMap = jet(my_handles.num+1); % Make a copy of jet.
% Assign white (all 1's) to black (the first row in myColorMap).
myColorMap(my_handles.num+1, :) = [1 1 1];
colormap(myColorMap); % Apply the colormap
colorbar
my_handles.xax = my_handles.uniquebldgs(:,1);
my_handles.yax = my_handles.uniquebldgs(:,2);

% --- Executes on button press in shapebldgreg.
function shapebldgreg_Callback(hObject, eventdata, handles)
global my_handles
my_handles.shapeflag = 1;

% --- Executes on button press in memHM.
function memHM_Callback(hObject, eventdata, handles)
if (get(handles.shapebldgreg,'Value')) == 1;
    global my_handles
    % Select .shp file to open
    [FileName,PathName] = uigetfile({'*.shp'},'Select Shapefile to
Load');
    entirepath = [PathName FileName];
    my_handles.TruthPolys = shape_get(entirepath);
    my_handles.fullpath = entirepath;
    [my_handles.shapeimage,my_handles.shapeallregs] =
ShapeBounds2Im(my_handles.TruthPolys,my_handles.HM);
    my_handles.numshape = length(my_handles.TruthPolys.Shape);
    % Plot
    axes(handles.axes2);
    cla reset
    imagesc(my_handles.shapeimage)
    xlabel('x Location (m)'), ylabel('y Location (m)'),
title(sprintf('%s%d','Building Regions (',my_handles.numshape,'
Total)'));
    axis xy
    axis equal
    myColorMap = jet(my_handles.numshape+1); % Make a copy of jet.
    % Assign white (all 1's) to black (the first row in myColorMap).
    myColorMap(my_handles.numshape+1, :) = [1 1 1];
    colormap(myColorMap); % Apply the colormap
    colorbar
    my_handles.xax = my_handles.shapeallregs(:,1);
    my_handles.yax = flipud(my_handles.shapeallregs(:,2));
end

% --- Executes on button press in loadHM.
function loadHM_Callback(hObject, eventdata, handles)
if (get(handles.shapebldgreg,'Value')) == 1;
    global my_handles
    % Select .shp file to open
    [FileName,PathName] = uigetfile({'*.shp'},'Select Shapefile to
Load');
    entirepath = [PathName FileName];
    my_handles.TruthPolys = shape_get(entirepath);
    my_handles.fullpath = entirepath;
    % Select HM to open
    [FileName,PathName] = uigetfile({'*.mat';'*.txt'},'Select Height
Model File to Load');
```

```
    entirepath = [PathName FileName];
    my_handles.HM = importdata(entirepath);
    [my_handles.shapeimage,my_handles.shapeallregs] =
ShapeBounds2Im(my_handles.TruthPolys,my_handles.HM);
    my_handles.numshape = length(my_handles.TruthPolys.Shape);
    % Plot
    axes(handles.axes2);
    cla reset
    imagesc(my_handles.shapeimage)
    xlabel('x Location (m)'), ylabel('y Location (m)'),
title(sprintf('%s%d','Building Regions (',my_handles.numshape,'
Total)'));
    axis xy
    axis equal
    myColorMap = jet(my_handles.numshape+1); % Make a copy of jet.
    % Assign white (all 1's) to black (the first row in myColorMap).
    myColorMap(my_handles.numshape+1, :) = [1 1 1];
    colormap(myColorMap); % Apply the colormap
    colorbar
    my_handles.xax = my_handles.shapeallregs(:,1);
    my_handles.yax = flipud(my_handles.shapeallregs(:,2));
end

function percentthresh_edit_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function percentthresh_edit_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in overalldamage.
function overalldamage_Callback(hObject, eventdata, handles)
global my_handles
my_handles.tilesize = str2double(get(handles.tilesize_edit,'String'));
my_handles.tilesizevar =
str2double(get(handles.tilesizevar_edit,'String'));
[my_handles.PercentDamage,my_handles.DamageMap] =
OverallDamageFINAL(my_handles.shapeflag);
PD = my_handles.PercentDamage(3,:)';
LowerDamagedPer = str2double(get(handles.percentthresh_edit,'String'));
% Generate Damage Map
DamMap = ones(size(my_handles.shapeimage));
map = PD > LowerDamagedPer;
for i = 1:length(map)
    if map(i) == 0
        DamMap(my_handles.shapeimage == i) = 2;
    else
        DamMap(my_handles.shapeimage == i) = 3;
    end
end
my_handles.DamMap = DamMap;
[l w] = size(my_handles.DamMap);
if l < 200 && w < 200
    my_handles.pos1 = 755 - w;
    my_handles.pos2 = 582 - l;
```

```
    my_handles.pos2bot = 181 - l;
    my_handles.pos3 = w*2;
    my_handles.pos4 = l*2;
else
    my_handles.pos1 = 755 - round(w/2);
    my_handles.pos2 = 582 - round(l/2);
    my_handles.pos2bot = 181 - round(l/2);
    my_handles.pos3 = w;
    my_handles.pos4 = l;
end
% Plot Initial Damage Asessment
axes(handles.axes2);
cla reset
set(handles.axes2,'Position',[my_handles.pos1,my_handles.pos2,my_handle
s.pos3,my_handles.pos4]);
imagesc(my_handles.xax,my_handles.yax,my_handles.DamMap)
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), title('Initial Damage Assessment');
axis xy

myColorMap(1, :) = [1 1 1];
myColorMap(2, :) = [0 0 0];
myColorMap(3, :) = [1 0 0];
colormap(myColorMap);
% Plot Site Image
[pathstr, my_handles.name, ext, versn] =
fileparts(my_handles.fullpath);
my_handles.siteImage = imread([my_handles.name,'.png']);
axes(handles.axes1);
cla reset
set(handles.axes1,'Position',[my_handles.pos1,my_handles.pos2bot,my_han
dles.pos3,my_handles.pos4]);
imagesc(my_handles.siteImage)
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), title([my_handles.name,' Site']);
axis off

% --- Executes on button press in initdammap.
function initdammap_Callback(hObject, eventdata, handles)
global my_handles
% Plot Initial Damage Asessment
axes(handles.axes2);
cla reset
set(handles.axes2,'Position',[my_handles.pos1,my_handles.pos2,my_handle
s.pos3,my_handles.pos4]);
imagesc(my_handles.xax,my_handles.yax,my_handles.DamMap)
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), title('Initial Damage Assessment');
axis xy
myColorMap(1, :) = [1 1 1];
myColorMap(2, :) = [0 0 0];
myColorMap(3, :) = [1 0 0];
colormap(myColorMap);

% --- Executes on button press in dampermap.
function dampermap_Callback(hObject, eventdata, handles)
global my_handles
```

```
axes(handles.axes2);
cla reset
set(handles.axes2,'Position',[my_handles.pos1,my_handles.pos2,my_handle
s.pos3,my_handles.pos4]);
imagesc(my_handles.xax,my_handles.yax,my_handles.DamageMap)
xlabel('Easting (UTM 18N x Coordinate)'), ylabel('Northing (UTM 18N y
Coordinate)'), title('Percent Damage Map');
axis xy
myColorMap = jet(100);
colormap(myColorMap);
colorbar;

% --- Executes on button press in resetpage.
function resetpage_Callback(hObject, eventdata, handles)
close(DamageDetectionFINAL)
eval('DamageDetectionFINAL')
```

TileNormDamage.m

```
function
[IndDamPts,OffNormals,OffXYZ,OnNormals,OnXYZ,Metric1,Metric2,UseableNum
Flag] = TileNormDamage(RegionTiles,Region)
% TileNormDamage computes the best fit plane for the set of points in
% each tile (using PCA), determines the normal vector for each plane,
% and compares the normal vectors by referencing to a zenith angle and
% looking for outliers.
%
% Rick Labiak
% Last Modified 11 August 2011

Normals = zeros(3,length(RegionTiles));
MeanXYZ = zeros(3,length(RegionTiles));
Errors = zeros(1,length(RegionTiles));
Points = [];
ExtraPtsIndex = [];
Residuals = cell(length(RegionTiles),1);
UseableNumFlag = 0;

for k = 1:length(RegionTiles)

    XYZ = RegionTiles{k,1};
    MeanXYZ(:,k) = mean(XYZ(:,1:3),1);

    if isempty(XYZ)
        ExtraPtsIndex = horzcat(ExtraPtsIndex,k);
    else
        if size(XYZ,1) < 3 || sum(isnan(XYZ(:,3)))>=1
            Points = vertcat(Points,XYZ);
            ExtraPtsIndex = horzcat(ExtraPtsIndex,k);
        else
            A = XYZ(:,1:3);
            [coeff,score,roots] = princomp(A);
            basis = coeff(:,1:2);
            normal = coeff(:,3);
            [n,p] = size(A);
```

```
            meanA = mean(A,1);
            Afit = repmat(meanA,n,1) + score(:,1:2)*coeff(:,1:2)';
            residuals = A - Afit;
            error = abs((A - repmat(meanA,n,1))*normal); % Also the
            norm of each residual
            sse = sum(error.^2);
            Normals(:,k) = normal;
            Errors(k) = sse;
            Residuals{k,1} = residuals;
        end
    end

end

normneg = Normals(3,:) < 0;
Normals(:,normneg) = Normals(:,normneg).*-1;

% Compare Normals to zenith angle
zenith = [0 0 1];
[m n] = size(Normals);
Zeniths = repmat(zenith',1,n);

Angle = acosd(dot(Zeniths,Normals));
Angle(Angle == 90) = nan;

% Determine the number of "useable" angles and if less than
UseableNum = sum(~isnan(Angle));
if UseableNum < 4 || size(Region,1)<50
    IndDamPts = Region;
    OffNormals = [];
    OffXYZ = [];
    OnNormals = [];
    OnXYZ = [];
    Metric1 = 0;
    Metric2 = 1;
    UseableNumFlag = 1;
else
    x = 1:1:90;
    nx = hist(Angle,x);
    emptybins = numel(find(nx == 0));
    Metric1 = emptybins/length(x);

    % Identify poor normals
    on_ind = [];
    y = 1:5:90;
    per = 0.2;
    thresh = per*UseableNum;
    [ny] = hist(Angle,y);
    ind = find(ny >= thresh & ny > 1);
    if isempty(ind)
    else
        for i=1:numel(ind)
            if ind(i) == 1
                tempon_ind = find(Angle < y(ind(i)+1)-(5/2));
            elseif ind(i) == length(y)
                tempon_ind = find(Angle > y(ind(i)-1)+(5/2));
            else
```

```
                tempon_ind = find(Angle > y(ind(i)-1)+(5/2) & Angle <
y(ind(i)+1)-(5/2));
            end
            on_ind = horzcat(on_ind,tempon_ind);
        end
    end
    on_ind = unique(on_ind);
    OnNormals = Normals(:,on_ind);
    OnXYZ = MeanXYZ(:,on_ind);
    off_ind = 1:n;
    off_ind(on_ind) = [];
    tf = ismember(off_ind,ExtraPtsIndex);
    off_ind = off_ind(~tf);
    OffNormals = Normals(:,off_ind);
    OffXYZ = MeanXYZ(:,off_ind);

    % Look at individual points
    % Call all points in "poor normal" tiles damaged
    IndDamPts = [];
    for i=1:length(off_ind)
        PoorNormalPtsTemp = RegionTiles{off_ind(i),1};
        IndDamPts = vertcat(IndDamPts,PoorNormalPtsTemp);
    end
    % Add ungrouped points
    IndDamPts = vertcat(IndDamPts,Points);

    Metric2 = size(OffNormals,2)/UseableNum;
end
```

DamDetVar.m

```
function [DamageList1,Metric] = DamDetVar(RegionTiles,Region)
% DamDetVar calculates the height variance in each tile to detect
% damage.
%
% Rick Labiak
% Last Modified 11 August 2011

Points = [];

% Initialize Damage List
DamageList1 = [];

Var = zeros(length(RegionTiles),1);

for k = 1:length(RegionTiles)

    XYZ = RegionTiles{k,1};

    if isempty(XYZ)
    elseif size(XYZ,1) < 3
        Points = vertcat(Points,XYZ);
    else
        Var(k) = std(XYZ(:,3))^2;
    end
```

```
end

UsefulVar = Var;
if isempty(UsefulVar)
    DamageList1 = Region;
    Metric = length(DamageList1)/length(Region);
else
    UsefulVar(:,2) = 1:size(UsefulVar,1);
    UsefulVar(Var == 0,:) = [];

    upperthreshout = 0.03;
    tileindextemp = UsefulVar(:,1) > upperthreshout;
    tileindex = UsefulVar(tileindextemp,2);

    for k = 1:length(tileindex)
        DamageTile = RegionTiles{tileindex(k),1};
        DamageList1 = vertcat(DamageList1,DamageTile);
    end
    % Add ungrouped points
    DamageList1 = vertcat(DamageList1,Points);
    Metric = length(DamageList1)/length(Region);
end
```

ShapeBounds2Im.m

```
function [I,AllRegions] = ShapeBounds2Im(TruthPolys,HM)
% Shape2BoundsIm takes in a set of polygon boundaries, finds the height
% model points within each boundary, and creates an image that can be
% displayed in the GUI.
%
% Rick Labiak
% Last Modified 11 August 2011

% Get entire list of points marked by region
AllRegions = [];
XYZH = HM;
for i = 1:length(TruthPolys.Shape);
    IN =
inpolygon(XYZH(:,1),XYZH(:,2),TruthPolys.Shape(i).x,TruthPolys.Shape(i)
.y);
    Region = XYZH(IN,:);
    TempRegion = horzcat(Region,ones(length(Region),1).*i);
    AllRegions = vertcat(AllRegions,TempRegion);
end

a = AllRegions;

% Grid points
res = 1;
maxX = max(a(:,1));
minX = min(a(:,1));
maxY = max(a(:,2));
minY = min(a(:,2));
xrange = maxX-minX;
```

```
yrange = maxY-minY;
% Subtract remainder from extent so entire area can be evenly divided
% into grids
Rx = mod(xrange,res);
Ry = mod(yrange,res);
xrange = xrange - Rx;
yrange = yrange - Ry;
% Sort points by X, then Y
SortedXYZ = sortrows(a,[1 2]);
image = zeros(yrange/res,xrange/res);
i = 0:res:yrange;
j = 0:res:xrange;
ytemp = i+minY;
xtemp = j+minX;

for m=1:length(i)-1
    for n=1:length(j)-1
        index = find(SortedXYZ(:,1) > xtemp(n) & SortedXYZ(:,1) <
xtemp(n+1) & SortedXYZ(:,2) > ytemp(m) & SortedXYZ(:,2) < ytemp(m+1));
        if isempty(index)
        else
            image(m,n) = max(SortedXYZ(index,8));
        end
        clear index
    end
end

% Clean up image
I = PixelFill(image,1,0);
I = PixelFill(I,1,1);
I(I == 0) = length(TruthPolys.Shape)+1;
I = round(I);
```

OverallDamageFINAL.m

```
function [PercentDamage,DamMap] = OverallDamageFINAL(shapeflag)
% OverallDamageFINAL calculates the percent damage for each building
% region in the scene and outputs a damage map.
%
% Rick Labiak
% Last Modified 11 August 2011

global my_handles
% Loop through each region (building) and determine percent damage
if shapeflag == 0
    % Initialize percent damage matrix
    PercentDamage = zeros(4,length(my_handles.Bounds));
    for i = 1:length(my_handles.Bounds)
        IN =
inpolygon(my_handles.HM(:,1),my_handles.HM(:,2),my_handles.Bounds{i}(:,
2),my_handles.Bounds{i}(:,1));
        my_handles.Region = my_handles.HM(IN,:);
        % Tile Normals
        [my_handles.RegionTiles,SortedRegion,m,n,tilecount] =
TileCloud(my_handles.Region,my_handles.tilesize);
```

```
[my_handles.IndDamPts,OffNormals,OffXYZ,OnNormals,OnXYZ,my_handles.Metr
ic1,my_handles.Metric2,UseableNumFlag] =
TileNormDamage(my_handles.RegionTiles,my_handles.Region);
        PercentDamage(1,i) = round(my_handles.Metric2*100);
        % Tile Variance
        [my_handles.RegionTiles,SortedRegion,m,n,tilecount] =
TileCloud(my_handles.Region,my_handles.tilesizevar);
        [my_handles.DamPtsTileVar,my_handles.Metric3] =
DamDetVar(my_handles.RegionTiles,my_handles.Region);
        PercentDamage(2,i) = round(my_handles.Metric3*100);
        % Rules
        % 2. Check to see if the number of useable normal angles if
        % less than 4 - if so variance is used
        if UseableNumFlag == 1
            PercentDamage(3,i) = PercentDamage(2,i);
        else
            PercentDamage(3,i) = PercentDamage(1,i);
        end
        % 1. Check to see if 90% of points are below 1 m - if so assign
        % 100% damage to building region
        thresh = 0.9*size(my_handles.Region,1);
        tempind = find(my_handles.Region(:,3)<1);
        if numel(tempind) > thresh
            PercentDamage(3,i) = 100;
        else
        end
        clear thresh tempind
    end
    % Generate Damage Map
    DamMap = zeros(size(my_handles.uniquebldgs));
    for i = 1:length(my_handles.Bounds)
        DamMap(my_handles.uniquebldgs == i) = PercentDamage(3,i);
    end
else
    % Initialize percent damage matrix
    PercentDamage = zeros(3,my_handles.numshape);

    for i = 1:my_handles.numshape
        ind = my_handles.shapeallregs(:,8) == i;
        my_handles.Region = my_handles.shapeallregs(ind,:);
        % Tile Normals
        [my_handles.RegionTiles,SortedRegion,m,n,tilecount] =
TileCloud(my_handles.Region,my_handles.tilesize);

[my_handles.IndDamPts,OffNormals,OffXYZ,OnNormals,OnXYZ,my_handles.Metr
ic1,my_handles.Metric2,UseableNumFlag] =
TileNormDamage(my_handles.RegionTiles,my_handles.Region);
        PercentDamage(1,i) = round(my_handles.Metric2*100);
        % Tile Variance
        [my_handles.RegionTiles,SortedRegion,m,n,tilecount] =
TileCloud(my_handles.Region,my_handles.tilesizevar);
        [my_handles.DamPtsTileVar,my_handles.Metric3] =
DamDetVar(my_handles.RegionTiles,my_handles.Region);
        PercentDamage(2,i) = round(my_handles.Metric3*100);
        % Rules
        % 2. Check to see if the number of useable normal angles if
```

```
        % less than 4 - if so variance is used
        if UseableNumFlag == 1
            PercentDamage(3,i) = PercentDamage(2,i);
        else
            PercentDamage(3,i) = PercentDamage(1,i);
        end
        % 1. Check to see if 90% of points are below 1 m - if so assign
        % 100% damage to building region
        thresh = 0.9*size(my_handles.Region,1);
        tempind = find(my_handles.Region(:,3)<1);
        if numel(tempind) > thresh
            PercentDamage(3,i) = 100;
        else
        end
        clear thresh tempind
    end

    % Generate Damage Map
    DamMap = zeros(size(my_handles.shapeimage));
    for i = 1:my_handles.numshape
        DamMap(my_handles.shapeimage == i) = PercentDamage(3,i);
    end
end
```